1/93
SONDERHEFT

AMIGA MAGAZIN

Markt&Technik

DM 12,-
öS 96,-
sfr 12,-

Disketten zum Heft als Public Domain. Mit allen Listings und vielen Extras

Ram Disk 100% full, 0K free, 3K in use

Faszination Programmieren

BASIC
C
Oberon

AMIGA PROGRAMMIEREN

Alles zur Programmierung des Amiga.
Themen dieser Ausgabe sind z.B.:

- Compiler-Bau
- 3-D-Effekte
- Besonderheiten unter OS 2.1 und 3.0
- Objektorientierte Programmierung
- Strategiespiel-Algorithmen

und viele Tips & Tricks, für alle, die programmieren oder es lernen wollen

Von Amiga-Experten geschrieben.
Mitwirkende Autoren u.a.:

- Peter Wollschlaeger
- Rudolf und Ilse Wolf
- Friedjof Siebert
- Edgar Meyzis
- Jens Gelhar
- Kai Bolay

und, und

GROSSER Programmierwettbewerb: Gewinnen Sie eine GVP-Turbokarte



Programmierer gesucht!





Seit mehr als 5 Jahren bieten wir Software für den Commodore Amiga an. In dieser Zeit haben wir über 150 kommerzielle Software-Produkte mit großem Erfolg veröffentlicht. Die Markenbezeichnung Schatztruhe haben wir zu einem Synonym für qualitativ hochwertige deutsche Produkte geprägt und Ihr Programm könnte bereits unser nächster Bestseller werden. Vertrauen Sie auf unser Know-How, und lassen Sie uns eine aktive Partnerschaft beginnen.



Wir suchen für die Computer der Commodore-Amiga-Familie ständig neu entwickelte Software für den kommerziellen Vertrieb: z.B. Büroanwendungen, Spiele, Grafiksoftware, DFÜ, DTP, Musik, kleinere Tools oder umfangreiche Projektentwicklungen.

Durch unsere regelmäßig erscheinenden Publikationen im Magazinbereich sowie durch die professionelle Schatztruhe-Serie in Verbindung mit einer optimalen Distributionspolitik, verfügen wir über die besten Voraussetzungen, Ihr Produkt bestmöglich zu vermarkten.



Wir bieten Ihnen Top-Konditionen, sowohl Festpreise als auch verkaufsabhängige Provisionen, und legen großen Wert auf eine intensive Zusammenarbeit mit ambitionierten Programmierern. Senden Sie uns eine Voll- oder Demoversion Ihrer Software ein. Herr Montenevoso, der Leiter unserer Abteilung für Projektplanung und Betreuung, wird sich kurzfristig mit Ihnen in Verbindung setzen.

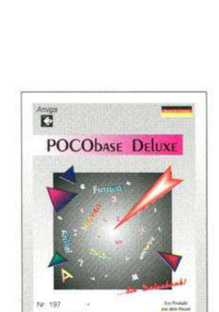


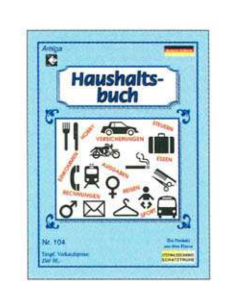

Fordern Sie außerdem kostenlos und unverbindlich unseren "Leitfaden für Programmierer" an, der Sie ausführlich über alle Aspekte des Vertriebes, Verdienstmöglichkeiten, Vertriebskonzepte u.ä. informiert.

Lassen Sie das Kapital in Ihrer Diskettenbox nicht verfallen. Senden Sie uns noch heute Ihre Software. Auch Hobby-Programmierer haben gute Chancen!

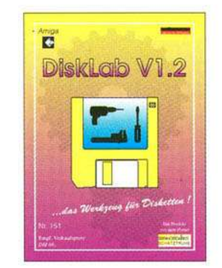
Wir freuen uns darauf, Ihre Software zu vermarkten! Setzen Sie sich also noch heute mit uns in Verbindung!



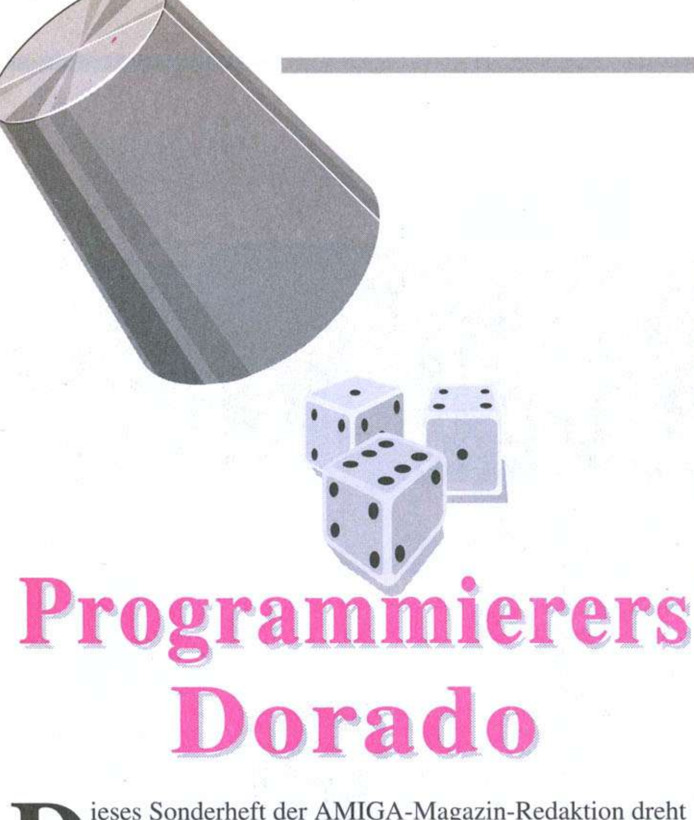








Stefan Ossowski´s Schatztruhe Gesellschaft für Software mbH, Veronikastraße 33, 4300 Essen 1, Tel.: 0201 / 79 84 47



Programmierers Dorado

Dieses Sonderheft der AMIGA-Magazin-Redaktion dreht sich ausschließlich ums Programmieren, und da ist bestimmt auch was für Sie dabei. Wir stellen ausführlich wichtige Programmiersprachen für den Amiga vor, richten uns mit Artikeln sowohl an Einsteiger als auch Profis. Unsere Autoren liefern Ihnen qualitativ hochwertige Informationen: Fridtjof Siebert (Entwickler von Oberon), Peter Wollschlaeger, Jens Gelhar (Vater des C++-Compilers), Thomas Pfrengle und Ulrich Sigmund (Cluster), Dietmar Heidrich (Entwickler des OMA-Assembler) u.a.

Für Programmierer ist der Amiga eine fantastische Spielwiese. Das belegt Monat für Monat das rege Interesse an den entsprechenden Rubriken des AMIGA-Magazins – auch ein Blick in den PD-Pool spricht Bände. Kein Wunder werden Sie sagen, verfügt der Amiga doch über ein Betriebssystem, das sich mit dem von Unix messen kann. Was Amiga-Programmierern schon in Fleisch und Blut übergegangen ist, bereitet den kommerziellen MS-DOS-Entwicklern Magengeschwüre: der Umstieg auf Windows. Die Umstellung auf Multitasking scheint größere Probleme zu bereiten als befürchtet. Uns, die es nicht anders kennen, bleibt das oftmals unverständlich. Grund genug, sich mit den Unterschieden der Windows- und Amiga-Programmierung auseinanderzusetzen. Unser Bericht »Amiga vs. Windows« verdeutlicht im Detail, welche Ungereimtheiten bei der Programm-Umsetzung vom Amiga auf Windows auftreten.

Besonders gespannt sind wir auf die Resonanz zu unserem Programmierwettbewerb (Seite 6). Ob er sich durchführen läßt, liegt einzig und allein an Ihnen – den Amiga-Programmierern und Ihrem Know-how.

Viel Spaß mit der ersten Ausgabe von »Faszination Programmieren« wünschen aus Ihrem AMIGA-Team:

Rainer Feith & Ulli Brieden

INHALT

Unsere Autoren stellen sich vor

Die Programmierer

Wer steht hinter der ersten Ausgabe des AMIGA-Magazin-Sonderhefts »Faszination Programmieren«? Natürlich eine ganze Menge Programmierer, die sich privat oder beruflich der Programmierung und speziell dem Amiga verschrieben haben.

von Ulrich Brieden

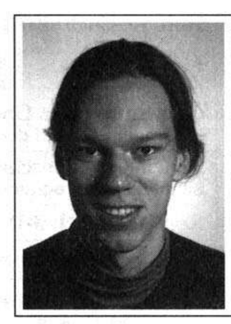
Damit Sie die Leute auch einmal kennenlernen, die an diesem Heft mitgewirkt haben, stellen wir Ihnen hier alle Autoren kurz vor.

Wie Sie selbst sehen, haben viele bekannte und berühmte Programmierer an diesem Sonderheft mitgewirkt. Experten wie Fridtjof Siebert, der Oberon-2 für den Amiga entwickelte, oder Peter Wollschlaeger, bekannter Autor diverser Fachbücher zum Amiga, waren genauso bereit mitzumachen, wie Edgar Meyzis, langjähriger Autor von Fachartikeln für das AMIGA-Magazin oder Ilse und Rudolf Wolf (in Österreich bekannt als die »Computerwölfe«).

Aber nicht nur die alten Hasen haben mitgemacht – auch die »Profis von Morgen« mischten mit. Am besten Sie schauen sich einfach mal an, wer alles dabei ist.

Und wenn Sie bei der nächsten Ausgabe mit anpacken möchten, schreiben Sie uns Ihre Vorschläge. Schicken Sie Ihre Beiträge einfach an unsere Redaktion:

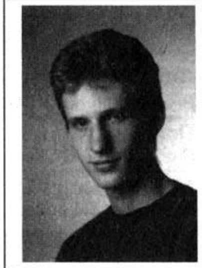
AMIGA-Magazin
Kennwort: Faszination Programmieren
Markt & Technik Verlag AG
Hans-Pinsel-Str. 2
8013 Haar bei München ■



Fridtjof Siebert: Wer kennt ihn nicht. Er ist der Entwickler von Oberon-2 für den Amiga. 1971 hat man ihn »auf diese Welt losgelassen«. Im dreizehnten Lebensjahr bekam er einen C 64, wechselte dann zum Amiga 500. Seit 1990 entwickelt er Oberon. Nebenbei studiert er Informatik an der Stuttgarter Uni, nachdem er seinen Zivildienst ableistete.



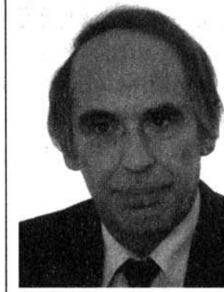
Dietmar Heidrich
Erste Computerschritte erlernte der jetzt 24jährige 1991 auf einem Apple II. Auch die Ära des VC-20 erlebte er mit. 1984 hatte er den ersten eigenen Computer, einen C 64 mit Floppy. Er programmierte kleinere Programme, darunter ein Disk-Checker und ein Kopierprogramm fürs DolphinDOS sowie den Debugger »Debug64«. 1986 lernte er den Amiga kennen und entwickelt seit 1988 den optimierenden Amiga-Makro-Assembler »OMA«. Seit Oktober 1989 studiert er Informatik an der Technischen Universität Karlsruhe.



Christof Brühann
Der 20jährige kam wie viele andere auch über den C 64 zum Amiga. Der Abiturient leistet z.Zt. seinen Zivildienst ab und wird noch in diesem Jahr sein Informatikstudium beginnen. Seinen Amiga nutzt er vorwiegend zum Programmieren in C und Assembler.



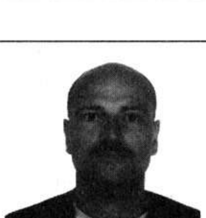
Edgar Meyzis
Den AMIGA-Magazin-Lesern ist er durch seine fantastischen Programmierkurse ein Begriff. Er studierte E-Technik, Wirtschaftswissenschaften und Informatik und ist z.Zt. als Berater nach mehrjähriger Leitung eines Ingenieurbüros tätig. Er programmiert den Amiga gerne wegen seines modernen Betriebssystems, das Experimente zuläßt, deren Ergebnisse auf größere Systeme übertragbar ist.



Peter Wollschlaeger
Er kennt den Amiga in- und auswendig. In Amiga-Kreisen wurde er bekannt durch das »Amiga Assembler-Buch«, das »C-Workshop-Buch« und dem Buch »Profi-Tips und Power-Tricks«. Doch auch auf anderen Computer-Systemen fühlt er sich heimisch – Bücher über Windows und Macintosh belegen es. Er hat mit den verschiedenen grafischen Oberflächen keinerlei Probleme: »Wer einmal gelernt hat, wie ein GUI (Graphic User Interface) eingesetzt wird, kann schnell wechseln. Die Denkweise ist dieselbe, nur die Syntax ist neu.«



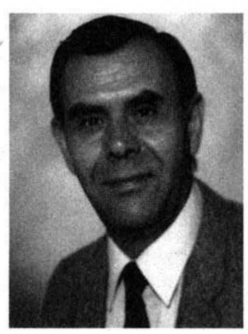
Astrid Brüggemann
Dort hinter der Kamera, die mich gerade fotografiert, steht mein Papa. Er heißt Bernfried. Wenn ich abends schlafe, spielt er manchmal mit seinen Computern. Am Tag darauf ist er immer ganz müde. Er ist sogar schon mal mittendrin beim Memory spielen auf dem Teppich eingeschlafen. Dann habe ich ihm ein Kissen unter den Kopf geschoben. So ist mein Papa.



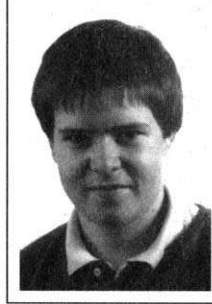
Bernhard Emese
Seitdem er den Taschenrechner HP-25 besitzt (49 Programmschritte) und darauf »Schiffe versenken« programmierte, hat ihn das Thema gefesselt. 1985 legte er sich einen Amiga 1000 zu. Ein Urlaub in Südfrankreich, der dem Zweck diente, ein ultimatives Computerspiel mit zwei weiteren Programmierern zu entwickeln, scheiterte zwar am schönen Wetter, hatte aber dennoch etwas Gutes: seitdem beherrscht er die C-Programmierung. Heute führt er zusammen mit einem der beiden Programmierer eine Firma für Film- und Motion-Control-Software.



Raphael Koch
Den ersten Kontakt mit einem Computer bekam er 1987. Es war ein C 64. Über BASIC kam er zur Programmiersprache Assembler. 1991 schließlich stieg er auf den Amiga um, programmierte in Amiga-BASIC, lernte C und beherrscht heute MC-68000-Assembler.

Ilse und Rudolf Wolf:
Den AMIGA-Magazin-Lesern sind beide als Österreich-Korrespondenten bekannt. Rudolf Wolf kam schon während seines Studiums an der TU-Wien mit Computern in Kontakt – dem »Mailüfterl«, der heute im Wiener Technischen Museum zu bewundern ist. Schon damals publizierte er erste Artikel in Elektronik-Zeitschriften und entwickelte Einplatinen-Computer. In seinem Fahrwasser entwickelte sich seine Frau Ilse ebenfalls zu einer EDV-Spezialistin und Computer-Journalistin und steht heute im Impressum von vier Fachzeitschriften.



Oliver Reiff
Seit geraumer Zeit ist auch er als freier Autor für das AMIGA-Magazin tätig. Sein Start in die Computerwelt begann 1987, natürlich mit einem Amiga. Er programmiert leidenschaftlich gern MC68000-Assembler. Der Noch-Zivildienst-leistende wird anschließend ein Informatikstudium in der Karlsruher oder Stuttgarter Universität beginnen.



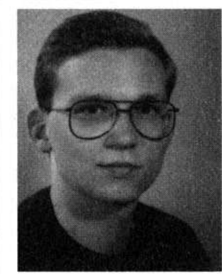
Thomas Pfrengle
Mehr oder weniger dem Zufall ist es zu verdanken, das sein Interesse am Computer weckte. Ein C 64 war 1984 sein erster Computer. Seit 1989 verfügt er über einen Amiga und ist neben Ulrich Sigmund maßgeblich an der Entwicklung der Programmiersprache »Cluster« beteiligt. Thomas Pfrengle ist Informatikstudent in Karlsruhe und arbeitet derzeit fieberhaft an einer neuen Cluster-Version.



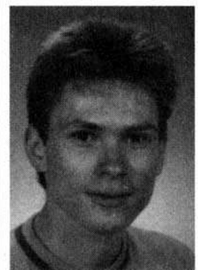
Alexander Kochann
Der 20jährige studiert z.Zt. an der Mannheimer Universität Wirtschaftsinformatik und kann schon jetzt auf eine über zehnjährige Computererfahrung zurückblicken. Sein erster Computer war der Atari VCS, dem der C 64 und der Amiga folgte. Sein Wissen stellt er schon seit einiger Zeit den Amiga-Fans als freier Autor des AMIGA-Magazins zur Verfügung.



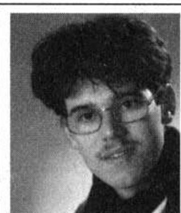
Sebastian Wedeniwski
Sein Steckenpferd ist die Grafikprogrammierung in allen Varianten. Für die Implementation verwendet er C und Pascal, sowohl auf dem Amiga als auch dem PC. Da er reges Interesse an der Mathematik hat, ist für ihn auch die Informatik von Bedeutung (z.B. Algorithmenentwicklung, Komplexitätstheorie und formale Sprachen). 1991 nahm er am Bundeswettbewerb für Informatik teil; 1992 erreichte er die Endrunde. Glückwunsch.



Daniel Görtz
Er begann nicht mit dem C 64, sondern dem Schneider CPC 464. Die weiteren Stationen waren schließlich doch der C 64, dann der Amiga. Er bevorzugt die Programmiersprache C, da in dieser Rubrik gute Compiler existieren und zudem sehr effizienter Code erzeugt wird.



Markus Öllinger
Erste Computerkenntnisse sammelte er mit dem Sinclair ZX.81. Bevor er sich endgültig für einen Amiga entschied, lernte er die Assembler-Programmierung auf dem C 64. Heute schreibt er seine Programme zum größten Teil in der Programmiersprache C, so auch das Programm des Monats vom AMIGA-Magazin 7/91. Markus Öllinger studiert Telematik an der Technischen Universität Graz.



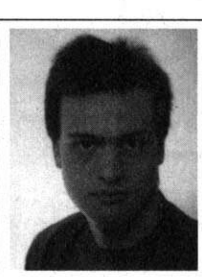
Kai Bolay
1973 erblickte er das Licht der Welt und absolvierte am Ende seiner schulischen Laufbahn sein Abitur, dem der Zivildienst folgte. Auch er fing mit dem legendären C 64 an. Der Umstieg auf den Amiga folgte zwangsläufig, der zunächst in BASIC und Assembler programmiert wurde. Vor drei Jahren wechselte Kai Bolay zu den Programmiersprachen Modula-2 und Oberon.



Stefan Herr
Der jetzt 22jährige studiert derzeit im siebten Semester Informatik an der Karlsruher Universität. Ersten Kontakt mit Computern hatte er schon 1979 mit dem legendären »PET«. Seit 1987 entwickelt er kommerzielle Software auf dem Amiga und ist z.Zt. mit der Erstellung der EGS-Dokumentation beschäftigt (Extended Graphics Standard).



Georg Herbold
Sein erster Computer war gleich der Amiga, den er sich 1987 zulegte. Erste Programmiererfahrungen sammelte er in Amiga-BASIC, stieg aber auf Assembler um und spezialisiert sich seitdem auf die Programmierung des Blitters und schneller Rechenoperationen. Zur Zeit studiert er Luft- und Raumfahrttechnik in Stuttgart.



Roger Fischlin
Zuerst waren es Video-Spiele, dann der C 64, dem schließlich der Amiga folgte. Der anfänglichen Spielewut folgte kurze Zeit später das Interesse am Programmieren. Da sich BASIC als zu langsam für die Spieleprogrammierung erwies, stieg er noch zu den C 64-Zeiten auf Assembler um. Dieser Sprache hält er auch heute noch die Treue, obwohl er mittlerweile die Vorteile von Hochsprachen schätzen lernte. Zur Zeit studiert er Informatik an der Frankfurter Universität.

Programmierer aufgepaßt:

Das AMIGA-Projekt

Wir nehmen dieses Sonderheft der AMIGA-Magazin-Redaktion für Programmierer als Forum und rufen zu einem Wettbewerb auf, der seinesgleichen sucht: Wir möchten mit Ihnen ein Programmierprojekt durchführen.

von Rainer Zeitler

Die Welt des Amiga verfügt über ein gewaltiges Potential exzellenter Programmierer. Wirft man einen Blick in den PD-Pool, merkt man: Gerade in Deutschland schlummern ungeahnte Talente im Verborgenen. Hunderte von namenlosen Einzelkämpfern sind in der Lage, fantastische Utilities zu konstruieren – ein größeres Projekt läßt sich alleine allerdings schwer realisieren. Genau hier setzen wir an. Wir möchten guten Programmierern eine Plattform bieten und mit entsprechender Koordination die Applikation entwickeln, die nur so möglich ist.

Die Idee war schnell geboren, lediglich die Umsetzung bereitete Kopfzerbrechen: Welches Projekt sollten wir in Angriff nehmen? Mit welcher Programmiersprache ist es durchzuführen? Wer plant und organisiert es?

Nach einigem Hin und Her entschieden wir uns für die Programmiersprache **»Oberon-2«**. Sicher, es gibt andere, ebenso leistungsfähige, z.B. »Cluster«, »C« oder »C++«. Obwohl C gerade im Amiga-Bereich eine beliebte Programmiersprache ist, zeigt sie doch gravierende Schwächen – C-Quelltext ist nur sehr schwer nachvollziehbar und für unser Projekt somit nicht geeignet. Gleiches gilt für C++. Blieb also noch Oberon bzw. Cluster. Beide bieten leistungsfähige Features und jede Sprache hat ihre speziellen Vorzüge. Einer davon sprach für Oberon-2: die leichte Portierbarkeit auf andere Betriebssysteme. Da Cluster eine Amiga-Implementation ist, entfällt die Umsetzung und ist somit für unser Projekt unzweckmäßig.

Die Programmiersprache stand also fest, doch ein weiteres Problem trat auf: Welche Applikation kommt in Frage? In jedem Fall muß sie der Bedingung genügen, daß die Planung auch mit mehreren Programmierern durchführbar ist. Drei Vorschläge von uns:

❐ Tabellenkalkulation
❐ Textverarbeitung
❐ Projekt-Management für verschiedene Programmiersprachen

Die ersten beiden Alternativen erklären sich selbst. Hinter dem Projekt-Management verbirgt sich eine für Programmierer hochinteressante Idee: Sie soll bei der Kreation neuer Programme hilfreich sein: u.a. verwaltet sie die verschiedenen Module, beinhaltet einen eigenen Editor, unterstützt das Erstellen von Flußdiagrammen etc.

Wie nun können Sie an dem **Wettbewerb** teilnehmen? Zunächst einmal sollten Sie ein erfahrener Programmierer sein. Dabei ist es völlig unerheblich, ob Sie lediglich eigene oder kommerzielle Programme entwerfen. Sie sollten hochsprachenerfahren sein, denn noch so gute Assemblerkenntnis ist nicht gleichbedeutend mit der Beherrschung eines Hochsprachen-Compilers wie Oberon-2. Alle eingehenden Einsendungen werden von uns überprüft, bewertet und ausgewählt. Doch auch die Leser, die selbst nicht beim Wettbewerb mitmachen möchten, sind aufgerufen, ihre bevorzugte Anwendung abzugeben.

Was gibt's zu gewinnen? Alle Einsendungen, seien es nun Vorschläge zum Projekt oder Angebote von Programmierern, die gerne Ihren Beitrag zu einem fantastischen Programm leisten möchten, kommen in einen großen Topf. Die Auslosung wird einen Gewinner bestimmen, der ein **GVP-Turboboard (MC68030-Prozessor, 25 MHz)** für den Amiga 2000 nach Hause nehmen darf, gestiftet vom offiziellen deutschen Distributor für **GVP-Produkte, der DTM GmbH**, Dreiherrenstein 6a, 6200 Wiesbaden. Den Namen des Gewinners werden wir im nächsten »Faszination Programmieren« veröffentlichen. Der Rechtsweg ist ausgeschlossen.

Die Durchführung des Projekts hängt von mehreren Faktoren ab. Zunächst müssen mindestens **20 Programmierer** bereit sein, mitzuwirken. Die Schweizer Firma **A+L AG,** Däderiz 61, 2540 Grenchen, stellt für alle Beteiligten kostenlos die benötigten Oberon-2-Systeme zur Verfügung. Stehen die Teilnehmer fest, geht's an die Planung, das »Software Engineering«. Jeder Programmierer erhält genaue Instruktionen. Hierzu wird ein Treffen veranstaltet. Die weitere Kommunikation wird via Modem durchgeführt.

Selbstverständlich ist Enthusiasmus gefragt, denn ein Honorar kann es voraussichtlich nicht geben. Dennoch, das Programm unterliegt anschließend dem Copyright aller Beteiligten und wird der **Amiga-Fan-Gemeinde zur Verfügung** gestellt.

Selbst **Prof. Wirth**, Schöpfer der Sprachen »Pascal«, »Modula-2« und »Oberon« fand Interesse an unserem Projekt. Aus zeitlichen Gründen war es ihm aber nicht möglich, das Projekt intensiver zu unterstützen.

Wollen Sie mitmachen? Schicken Sie Ihre Vorschläge oder Unterlagen bis zum 14.2.1993 an:

AMIGA-Redaktion
Kennwort: Faszination Programmieren
Markt & Technik Verlag AG
Hans-Pinsel-Str. 2, 8013 Haar

Viel Erfolg wünscht Ihr Amiga-Team. ■

Programmieren auf der Workbench

Geheimnisvolles OS 2.0

Die Flexibilität der neuen Workbench hat Ihren Preis. Konnte man unter früheren Kickstart-Versionen nur bedingt Änderungen an der Workbench vornehmen, muß man als Programmierer seit OS 2.0 mit den ausgefallensten Voreinstellungen rechnen. Wir zeigen, was zu beachten ist.

von Alexander Kochann und Oliver Reiff

Es war einmal eine blaue Workbench mit weißer Schrift, dem fixen Zeichensatz Topaz 8 oder 9, einer Auflösung von 640 x 256 Pixeln und vier Farben. Und damit der Programmierer nicht allzuviel Arbeit hatte, gab's keinerlei Variationen. Man traf die Konfiguration auf sämtlichen Amiga-Modellen an – lediglich auf den Interlace-Modus oder die PAL/NTSC-Auflösung mußte man achten.

Aus und vorbei: Dank der vielfältigen Möglichkeiten der neuen Preferences-Programme finden sich immer mehr Anwender, die ihrer Kreativität freien Lauf lassen und ihre Workbench individuell gestalten. Selbstverständlich ist das eine tolle Errungenschaft des neuen Betriebssystems und nach Kräften zu unterstützen. Dennoch – Leidtragende dieser Einstellungsvielfalt sind die Programmierer, deren Programme auf alle Eventualitäten vorbereitet sein müssen. War es bisher unproblematisch, ein Workbench-Fenster zu öffnen, sollte man sich jetzt bereits vor dem Öffnen alle wichtigen Daten der Workbench besorgen. Dazu zählen u.a. der verwendete Zeichensatz und dessen Größe, die Bildschirmauflösung, die Anzahl der Farben usw.

Unser Assembler-Listing demonstriert die wichtigsten Merkmale, die zu beachten sind: Es öffnet ein Fenster auf der Workbench, gibt zwei Zeilen Text aus und wartet darauf, daß der Benutzer den Menüpunkt »Verlassen« auswählt oder einfach das Schließsymbol aktiviert. Eigentlich kein Problem – vor allem nicht unter OS 2.0 oder höher.

Schritt für Schritt

Unsere erste Handlung ist es, das Programm beim Nichtvorhandensein des neuen Betriebssystems abzubrechen. Wir vergleichen dabei die Version der Exec-Library mit der Zahl 37. Diese ist gleichbedeutend mit der von Commodore ausgelieferten Kickstart 2.04. Zwar geistern hier und da noch V36-Versionen umher – dabei handelt es sich aber um ehemalige Beta-Versionen; sie sollten nicht berücksichtigt werden.

Ab jetzt gilt's. Zunächst ist die Adresse des Workbench-Screens in Erfahrung zu bringen. Über diesen Screen-Zeiger läßt sich allerhand Interessantes in Erfahrung bringen; doch dazu später mehr. Die Intuition-Library (ebenfalls V37) stellt hierfür die Funktion LockPubScreen() zur Verfügung. Diese rufen wir einfach mit dem Parameter »Workbench« auf. Gegenüber der schon bekannten OpenWorkbench()-Funktion besitzt sie einen entscheidenden Vorteil: Der Screen (in unserem Fall ist das der Workbench-Schirm) läßt sich nicht mehr schließen, es sei denn, wir benutzen UnlockPubScreen(). Dieser Befehl teilt dem Betriebssystem bzw. dem Programm, das den Schirm öffnete, mit, daß wir ihn nicht mehr benötigen. Tätigen wir den Aufruf nicht, ist es bis zum Neustart nicht mehr möglich, den Schirm zu schließen. Also nicht vergessen!

Bevor wir unser Fenster mit OpenWindow() öffnen, benötigen wir noch das eine oder andere Datum, z.B. die Breite und Höhe der Workbench. Da ein Fenster aber bekanntlich (bis auf eine Ausnahme) mitsamt Rahmen dargestellt wird, sind eigentlich die Rahmenkoordinaten interessant, denn die Höhe der Titelleiste ist variabel – eine einfache Zeichensatzänderung kann das bewirken.

Zum Glück spendiert uns das Betriebssystem die Tags »WA_InnerWidth« (TAG_USER+118) und »WA_InnerHeight« (TAG_USER+119). Mit diesen kann man die Höhe und Breite des Fensters ohne den Rahmen angeben, dessen Breite und Höhe das Betriebssystem dann für uns addiert (nicht zu verwechseln mit dem Window-Flag »Gim-

```
 1:           ****************************************************
 2:           *                WB-Beispielprogramm               *
 3:           *    von Alexander Kochann und Oliver Reiff        *
 4:           *                für DevPacII                      *
 5:           *            nur von der Shell starten             *
 6:           ****************************************************
 7:
 8: TAG_USER  equ      $80000000
 9:
10:           opt      a+,o+,p+
11: Start     move.l   4.w,a6
12:           lea      IntName,a1
13:           moveq    #36,d0             * OS2.x oder höher
14:           jsr      -552(a6)           OpenLib
15:           lea      IntBase,a0
16:           move.l   d0,(a0)
17:           beq      No_OS2             * Kein OS2.x
18:           lea      GfxName,a1
19:           jsr      -408(a6)           OldOpenLib
20:           lea      GfxBase,a0
21:           move.l   d0,(a0)
22:           lea      GadName,a1
23:           jsr      -408(a6)           OldOpenLib
24:           lea      GadBase,a0
25:           move.l   d0,(a0)
26:           move.l   IntBase,a6
27:           lea      PupScreen,a0
28:           jsr      -510(a6)           LockPupScreen
29:           lea      Screen,a0
30:           move.l   d0,(a0)
31:           beq      No_Screen          * Kein Screen
32:           move.l   d0,a0
33:           jsr      -690(a6)           GetScreenDrawInfo
34:           lea      DrawInfo,a0
35:           move.l   d0,(a0)
36:           move.l   d0,a1
37:           move.l   8(a1),a1
38:           lea      Font,a0
39:           move.l   a1,(a0)            * dri_Font
40:           lea      YSize,a0
41:           move.w   20(a1),(a0)        * tf_YSize
42:           lea      BaseLine,a0
43:           move.w   26(a1),(a0)        * tf_BaseLine
44:           move.w   YSize,d0
45:           lsl.w    #1,d0              * mal 2
46:           addq.w   #8,d0              * plus 2 mal 4 Pixel
47:           lea      WA_InnerHeight+6,a0
48:           move.w   d0,(a0)            * Innere Höhe
49:           lea      Text_1,a0
50:           bsr      GetTextLenght
51:           move.l   d0,d1              * Pixelbreite Text 1
52:           lea      Text_2,a0
53:           bsr      GetTextLenght
54:           cmp.w    d0,d1              * Welcher Text ist breiter ?
55:           blt.s    .Width_Ok
56:           move.w   d1,d0
57: .Width_Ok addq.w   #8,d0              * plus 2 mal 4 Pixel
58:           lea      WA_InnerWidth+6,a0
59:           move.w   d0,(a0)            * Innere Breite
60:           move.l   Screen,a1
61:           lea      WindowData,a0
62:           move.b   30(a1),3(a0)       * TopEdge
63:           addq.b   #1,3(a0)
64:           lea      WindowTags,a1
65:           jsr      -606(a6)           OpenWindowTagList
66:           lea      Window,a0
67:           move.l   d0,(a0)
68:           beq      No_Window
69:           move.l   d0,a1
70:           lea      RastPort,a0
71:           move.l   50(a1),(a0)        * RastPort des Fensters
72:           lea      MessagePort,a0
73:           move.l   86(a1),(a0)        * MsgPort des Fensters
74:           move.l   GadBase,a6         * Menu einrichten
75:           lea      NewMenu,a0
76:           lea      NewMenuTags,a1
77:           jsr      -48(a6)            CreateMenusA
78:           lea      Menu,a0
79:           move.l   d0,(a0)
80:           beq      No_Menu
81:           move.l   Screen,a0
82:           sub.l    a1,a1
```

meZeroZero«). Dadurch wird das gesamte Fenster natürlich größer und ob es dann noch auf die Workbench paßt, hängt eben von deren Größe ab, die sehr stark variieren kann. Wir brauchen jetzt aber nicht umständlich die Höhe und Breite des Screens und des Fensters zu vergleichen. Es gibt in der Regel nur zwei Möglichkeiten, die einen Programmierer interessieren:

1. Das Fenster muß unbedingt in der angegebenen Größe geöffnet werden. Ist dies nicht möglich, bekommt man von OpenWindowTagList() eben eine Fehlermeldung (NULL) zurück. Jetzt sollte man den Benutzer mit einem Requester darauf aufmerksam machen, daß der Screen zu klein oder das Fenster zu groß ist.

2. Die Größe des Fensters kann im Notfall korrigiert werden. Dazu gibt man beim Aufruf noch den Tag »WA_AutoAdjust« (TAG_USER+144) an. Jetzt wird das Fenster so verschoben und verkleinert, daß es immer auf den Screen paßt. Diese Möglichkeit nutzen wir in unserem Beispielprogamm.

Jetzt möchten wir zwei Zeilen Text ausgeben. Dazu benötigen wir wichtige Informationen, und zwar z.B. die Höhe des eingestellten Zeichensatzes (Font). Wir bedienen uns hierzu der Funktion GetScreenDrawInfo(), die uns die Adresse des Zeichensatzes und andere wichtige Daten zur Verfügung stellt, die wir später noch benötigen. Im Element »dri_Font« (Offset 8) finden wir den Zeiger auf den voreingestellten Screen-Font. Das Element »tf_YSize« (Offset 20) enthält den von uns gesuchten Wert: die maximale Höhe des Zeichensatzes. Damit der Text nicht zu nahe am Rahmen plaziert wird, addieren wir jeweils vier Pixel Abstand oben und unten dazu.

Da wir die Fensterbreite der des Texts anpassen möchten, müssen wir also zunächst die erforderliche Textbreite in Erfahrung bringen. Hier hilft uns die Funktion »TextLength()« der Graphics-Library weiter. Die Funktion verlangt u.a. als Argument den Zeiger auf den RastPort unseres Fensters. Da wir den eingestellten Font benutzen, reicht es aus, den in der Screen-Struktur vorhandenen zu nehmen (»sc_RastPort«, Offset 84). Zur Breite addieren wir wiederum vier Pixel links und rechts.

Nun sollten wir uns die Position des Fensters überlegen. Viele Programme öffnen ihr Fenster direkt unter der Screen-Leiste. Deren Höhe finden wir in der Screen-Struktur im Element »sc_BarHeight« (Offset 30). Wir addieren einen Pixel und erhalten so die obere Position; die linke setzen wir auf Null.

Nachdem wir nun die wichtigsten Daten in Erfahrung gebracht haben, öffnen wir das Fenster mit der Funktion »OpenWindowTagList()«. Tritt ein Fehler auf, ist der Screen zu klein. Ist alles in Ordnung, läßt sich das erforderliche Menü erstellen und einbinden.

Auch hier kennt das Betriebssystem OS 2.0 leistungsfähige Routinen, die uns die meiste Arbeit abnehmen: wir finden sie in der Gadtools-Library. Zuerst müssen wir die Funktion CreateMenusA() mit der eigenen NewMenu-Struktur und einer optionalen TagList aufrufen. Das so erhaltene Menügerüst ist noch mit den Positionsdaten zu füllen. Das übernimmt die Funktion LayoutMenusA(), die allerdings noch einen sog. Zeiger auf eine VisualInfo-Struktur erwartet. Diesen erhalten wir durch Aufruf von GetVisualInfoA(). Ist auch das fehlerfrei geschehen, läßt sich die Menü-Struktur via SetMenuStrip() anfügen. Unabhängig vom eingestellten Zeichensatz stellt das Betriebssystem das Menü immer korrekt dar.

Zum Text: Den ersten möchten wir in Fettschrift ausgeben, den zweiten ohne jegliche Attribute. Nun darf man in keinem Fall davon ausgehen, daß Farbe 1 die Schrift- und Farbe 0 die Hintergrundfarbe ist – ab OS 3.0 läßt sich auch das vom Benutzer variabel einstellen. Welche Farbe welche Funktion erfüllt, erfahren wir aus der PenArray-Struktur. Das dri_Pens-Element (Offset 4) der DrawInfo-Struktur gibt darüber Auskunft.

Somit kennen wir jetzt auch die Textfarben. Was uns nun noch fehlt, ist die Textposition. Wir öffnen bewußt kein GimmeZeroZero-Fenster, da es wesentlich mehr Speicher benötigt und zudem extrem träge ist. Wir müssen uns demzufolge zunächst die Breite des linken und oberen Fensterrahmens besorgen: in den Elementen wd_BorderLeft (Offset 54) und wd_BorderTop (Offset 55) der Fensterstruktur. Vergessen dürfen wir nicht die vier Pixel Abstand vom Rahmen. Addieren müssen wir außerdem den Wert der Grundlinie des Zeichensatzes: aus dem Element tf_BaseLine (Offset 26) der Font-Struktur. Der Text kann ausgegeben werden.

Fertig? Nicht ganz. Nachdem wir die Nachricht (Message) zum Beenden erhalten haben, ist unbedingt alles freizugeben, was zuvor reserviert wurde: Die Menü-Struktur mit FreeMenus(), die VisualInfo-Struktur mit FreeVisualInfo(), FreeDrawInfo() und UnlockPupScreen(). Schießlich ist noch das Fenster zu schließen, auch die Libraries dürfen wir nicht vergessen.

Das Beispielprogramm zeigt einen Teil der Probleme auf und demonstriert, was unter OS 2.0 und höher u.a. zu beachten zu ist – aber eben nur einen Teil. Wer sicherstellen möchte, daß seine Programme kompatibel sind, sollte sich an die auf Seite 87 vorgestellten Grundregeln halten. *rz*

Literaturhinweise:

[1] Zeitler, Rainer: »Was lange währt ...« – Programmieren unter OS 2.0, AMIGA-Magazin 1-8/92, Markt & Technik Verlag AG

```
 83:            jsr       -126(a6)            GetVisualInfoA
 84:            lea       VisualInfo,a0
 85:            move.l    d0,(a0)
 86:            move.l    Menu,a0
 87:            move.l    VisualInfo,a1
 88:            lea       NewMenuTags,a2
 89:            jsr       -66(a6)             LayoutMenusA
 90:            tst.l     d0
 91:            beq       No_Layout
 92:            move.l    IntBase,a6
 93:            move.l    Window,a0
 94:            move.l    Menu,a1
 95:            jsr       -264(a6)            SetMenuStrip
 96:            move.l    Font,a0             * Font setzen und Text ausgeben
 97:            move.l    RastPort,a1
 98:            move.l    GfxBase,a6
 99:            jsr       -66(a6)             SetFont
100:            move.l    Window,a1
101:            moveq     #0,d0               * Text X-Pos
102:            moveq     #0,d1               * Text Y-Pos
103:            move.b    54(a1),d0           * LeftBorder
104:            move.b    55(a1),d1           * TopBorder
105:            add.w     BaseLine,d1         * plus Font Basislinie
106:            addq.w    #4,d0               * plus 4 Pixel
107:            addq.w    #4,d1               * plus 4 Pixel
108:            bsr       Move
109:            move.l    DrawInfo,a2
110:            move.l    4(a2),a2            * dri_Pens
111:            move.w    16(a2),d2           * hilighttextPen
112:            bsr       SetPen
113:            lea       Text_1,a0
114:            bsr       PrintText           * Text 1 hervorheben
115:            add.w     YSize,d1            * Nächste Zeile
116:            bsr       Move
117:            move.w    4(a2),d2            * textPen
118:            bsr       SetPen
119:            lea       Text_2,a0
120:            bsr       PrintText           * Text 2 normal
121: Wait       move.l    4.w,a6              * Wait wegen Multitasking
122:            moveq     #-1,d0
123:            jsr       -318(a6)            Wait
124: .NextMsg   move.l    MessagePort,a0
125:            jsr       -372(a6)            GetMsg
126:            tst.l     d0                  * keine Message
127:            beq.s     Wait
128:            move.l    d0,a1
129:            move.l    20(a1),d2           * kein Ereignis
130:            move.w    24(a1),d3           * Menupunkt
131:            jsr       -378(a6)            ReplyMsg
132:            cmp.l     #$200,d2            * CloseGadget ?
133:            beq.s     Ende
134:            cmp.l     #$100,d2            * Menu ?
135:            bne.s     .NextMsg
136:            cmp.b     #$40,d3             * 3. Punkt im 1. Menu ?
137:            bne.s     .NextMsg
138:
139: Ende       moveq     #0,d7               * Ok, kein Fehler
140: Err_Layout move.l    GadBase,a6          * Alles freigeben...
141:            move.l    Menu,a0
142:            jsr       -54(a6)             FreeMenus
143:            move.l    VisualInfo,a0
144:            jsr       -132(a6)            FreeVisualInfo
145: Err_Menu   move.l    IntBase,a6
146:            move.l    Window,a0
147:            jsr       -72(a6)             CloseWindow
148: Err_Window move.l    IntBase,a6
149:            move.l    Screen,a0
150:            move.l    DrawInfo,a1
151:            jsr       -696(a6)            FreeDrawInfo
152:            sub.l     a0,a0
153:            move.l    Screen,a1
154:            jsr       -516(a6)            UnlockPubScreen
155: Err_Screen move.l    4.w,a6
156:            move.l    GadBase,a1
157:            jsr       -414(a6)
158:            move.l    GfxBase,a1
159:            jsr       -414(a6)
160:            move.l    IntBase,a1
161:            jsr       -414(a6)
162: Err_OS2    move.l    d7,d0               * Fehlercode
163:            rts
164: *-----------------------------------------
```

Listing (Anfang); **Fortsetzung Seite 99**

O.M.A. 2.0

Das umfangreiche Makro-Assembler-Paket für alle Amiga Computer

Hier ist Ihr leistungsfähiges und schnelles Makro-Assemblerpaket. Für hochoptimierte Codes der Motorola-68000-Familie. Mit allem, was dazu gehört:
Der Editor hat 20 Textpuffer. Da finden Ihre Quellprogramme reichlich Platz.
Der Debugger öffnet Ihnen per Mausklick beliebig viele Fenster. Und damit Sie nicht den Durchblick verlieren, protokoliert er alle ausgeführten Befehle mit.
Der Linker fügt Ihre Module blitzschnell zu fertigen Programmen zusammen.

Weitere Hilfsprogramme, Includefiles und eine Assembler-Bibliothek sorgen für rasantes Tempo und komfortable Bedienung. Übrigens: O.M.A. 2.0 arbeitet auch mit 32-Bit-Prozessoren, Kickstart 2.0 und ECS. Und wenn Sie große, modularisierte Projekte verwalten müssen, rufen Sie einfach das Make-Utility auf.
O.M.A. hat eben wirlich alles, was dazu gehört.

Systemanforderungen:
Amiga 500, 1000, 2000, 3000 mit mindestens 512 Kbyte RAM Kickstart 1.2, 1.3, OS 2.0

Bestell-Nr. 500 85 **DM 198,95***
*unverbindliche Preisempfehlung

AMIGA-TEST
sehr gut

20 06 09



EIN UNTERNEHMEN VON



Schnittstellenprogrammierung

Serielle Machenschaften

von Bernhard Emese

Wollten Sie schon immer wissen, wie man die serielle Schnittstelle des Amiga effizient ansteuert und zusätzlich Übertragungsraten bis zu 38 400 Bit/s erreicht? Wir zeigen, mit welchen Tricks zu arbeiten ist und stellen eine funktionsfähige Implementation vor.

Mit der Zeit hat der Amiga schon so manches Terminalprogramm verkraften dürfen müssen. Doch statt zuverlässiger Datenempfänger sind daraus oft häßliche lahme und meist sogar noch vergeßliche Zeitgenossen geworden, indem sie Bytes verschluckten, die jeder andere Rechner noch locker aufgenommen hätte. Der Grund dafür ist das serielle Device des Amiga-Betriebssystems, das bei hohen Baudraten kneift und lieber ab 9 600 Bit/s Augen und vor allem Ohren zumacht, wenn es eigentlich gilt, Daten zu empfangen.

Die Hardware der Custom-Chips unterstützen die serielle Schnittstelle leidlich, so daß beispielsweise die Parity-Prüfung jedes ankommenden Bytes via Software (man beachte) Bit für Bit vorzunehmen ist. Kein Wunder also, daß die Zeit für hohe Übertragungsraten nicht ausreicht. Zwar entwickelte die damalige Commodore-Crew einen »superschnellen« Modus, den RAD_BOOGIE-Modus, vorrangig für MIDI-Ansteuerungen gedacht. Doch leider versagt auch dieser, wenn er 38 400 Bit/s zu empfangen hat.

Damit diese marode Prozedur auf die Beine kommt – hier finden Sie das Programm »Fastserial.c«. Dabei handelt es sich um einen kompletten interruptgesteuerten Handler zum Empfang serieller Daten auf dem Amiga, der rigoros auf Geschwindigkeit getrimmt ist und theoretisch noch mehr als 38 400 Bit/s empfangen könnte. Hier allerdings liegt bei der Hardware die Sende- und Empfangsgrenze. Es ist durchaus möglich, das Programm in ein Terminalprogramm einzubinden und so eine zuverlässige und bequem anzusprechende serielle Schnittstelle zu haben.

Anstatt die im Amiga-ROM-Kernel-Reference-Manual [1] beschriebene Prozedur des Allokierens von Ports, Extended Serial-IOs und OpenDevice-Aufrufen durchzuführen, genügt hier ein Aufruf von OpenSer() mit entsprechenden Parametern, und schon ist die Schnittstelle verfügbar.

Ein Minimalprogramm, das alle Funktionen des Handlers benutzt, einen Text sendet und anschließend einen anderen empfängt, zeigt das Listing. Über den IO-Pointer kann zusätzlich auch direkt auf die Kontrollstruktur für den seriellen Schnittstellentreiber zugegriffen werden. Dieselbe Struktur wird vom seriellen Device des Amiga in nahezu identischer Weise verwendet. Fastserial benutzt lediglich nicht alle Felder, sondern:

- io_Baud: Hier finden wir die aktuelle Baudrate
- io_RBufLen: Die Größe des Ringpuffers in Bytes
- io_ReadLen: Anzahl der Datenbits für den Empfang
- io_WriteLen: Anzahl der Datenbits fürs Senden
- io_StopBits: Anzahl Stopbits (1 oder 2)
- io_SerFlags: Ausgewertet wird nur mit SERF_PARTY_ON und SERF_PARTY_ODD

Die Struktur IOExtSer finden wir in der Include-Datei »devices/serial.h«:

```
struct  IOExtSer {
  struct  IOStdReq IOSer;
  ULONG   io_CtlChar;
  ULONG   io_RBufLen;
  ULONG   io_ExtFlags;
  ULONG   io_Baud;
  ULONG   io_BrkTime;
  struct  IOTArray io_TermArray;
  UBYTE   io_ReadLen;
  UBYTE   io_WriteLen;
  UBYTE   io_StopBits;
  UBYTE   io_SerFlags;
  UWORD   io_Status;
 };
```

Zusätzlich zu den aufgeführten Feldern trägt OpenSer() einen Pointer ins Feld IOExtSer.IOSer.io_Device ein. Der zeigt auf die Static-Variable RBFData, in der wir die Ringpufferadresse, die Anzahl der darin enthaltenen und noch nicht ausgelesenen Bytes und die aktuellen In- und Out-Counter finden.

```
 1: /*
 2:  * Amiga-Serial-Interrupt-Treiber für hohe Baudraten.
 3:  * Autor: Bernhard Emese
 4:  * Geschrieben für Aztec 5.0b ANSI C  5.1992
 5:  * Der Treiber ermöglicht Übertragungsraten bis zu
 6:  * 38400 Bit/s ohne Handshake. Beim Empfang wird dabei
 7:  * allerdings auf die Parity-Prüfung verzichtet, da
 8:  * sie per Software durchgeführt werden müßte und zuviel
 9:  * Zeit benötigt. Beim Senden wird das Parity-Bit jedoch
10:  * berechnet und gesendet, wenn Parity Enabled ist.
11:  * Compiler-Aufruf für Aztec C 5.0 (16-Bit-Modell)
12:  * cc fastserial.c -ps
13:  * Das Objekt-Modul kann mit C-Programmen zusammengebunden
14:  * werden, um so serielle Daten zu mit hoher Übertragungs-
15:  * rate zu empfangen.
16:  */
17: #include <exec/types.h>
18: #include <exec/memory.h>
19: #include <exec/interrupts.h>
20: #include <functions.h>
21: #include <hardware/custom.h>
22: #include <hardware/intbits.h>
23: #include <devices/serial.h>
24:
25: /*** Definitionen ***/
26: #define BAUDFAKTOR 3546895L   /* Quarz 28.37516 MHz / 8 */
27: #define SERINTNAME "fastserial.int"
28: #define OVRUN  0x8000           /* Overrun Bit */
29: #define RBF    0x4000        /* Receiver Buffer Full */
30: #define TBE    0x2000        /* TRansmitter Buffer empty */
31: #define TSRE   0x1000        /* Transmitter Register Empty */
32: #define RXD    0x0800
33: #define UARTLONG  0x8000  /*für neuntes Bit, wenn 8 Daten
34:                           und Parity gesendet werden soll */
35: /* Hardwareregister */
36: #define custom (*((struct Custom *)0xdff000))
37: /* Assembler Interrupt Handler als extern deklarieren */
38: extern void SerialReceiveInt();
39:
40: /*** Variablen ***/
41: /* interne Struktur RBFData, um den interruptgesteuerten
42:  * Receive Ringpuffer zu verwalten */
43: static struct SerialIntData {
44:   struct IOExtSer *io;
45:   UBYTE  *InCount,*OutCount;
46:   UBYTE  *Buffer;
47:   SHORT  Inhalt;
48: } RBFData;
49:
50: static struct Interrupt *SerInt=NULL,*PriorInterrupt=NULL;
51:
52: /*** Prototypes ***/
53: struct IOExtSer *OpenSer(ULONG Baud,ULONG BufLen,UBYTE
54: ReadLen,UBYTE WriteLen,UBYTE Stop,UBYTE Flags);
55: void   ParamSer(struct IOExtSer *io,ULONG Baud,ULONG
56: BufLen,UBYTE ReadLen,UBYTE WriteLen,UBYTE Stop,UBYTE Flags);
57: void   CloseSer(struct IOExtSer *io);
58: void   WriteSer(register struct IOExtSer *io,
```

Damit ist es dem Hauptprogramm möglich, den Puffer beliebig zu manipulieren, z.B. ihn zu löschen, indem die Ring-Counter auf die Pufferstartadresse gesetzt werden und der Inhaltzähler auf 0. Das ist z.B. dann sinnvoll, wenn man ein Abbruchzeichen empfängt, das alle zuvor gesendeten Daten unwirksam macht oder die Baudrate falsch eingestellt war und sich Datenmüll im Puffer befindet. Das Löschen läßt sich allerdings auch durch Auslesen mit ReadSer() ermöglichen, wobei letzteres aber wegen des Timeouts und der Schleife ein paar Millisekunden länger dauert. Es ist darauf zu achten, daß beim Zugriff auf RBFData ein Disable()-Befehl voran geht, da ansonsten eventuell eintreffende Zeichen verlorengehen, Das Löschen des Ringpuffers über RBFData muß deshalb so geschehen:

```
struct IOExtSer *io;
struct RBFData  *rbf;
/* io ist durch OpenSer() bereits
 * ordnungsgemäß initialisiert
 */
rbf=(struct RBFData *)io->IOSer.io_Device;
/* rbf zeigt auf RBFData */
Disable();
rbf->InCount=rbf->OutCount=rbf->Buffer;
/* Zähler auf Anfang des Puffers */
rbf->Inhalt=0;
Enable();
```

Diese Programmversion stellt natürlich nur einen Spezialfall dar: Den des schnellen Datenempfangs und -sendens. Eine allgemeingültigere Routine ließe sich jedoch sehr einfach erweitern, indem das XON/XOFF-Protokoll und eine Semaphore für die exklusive Nutzung verschiedener Tasks hinzugefügt werden. Vermutlich würde sogar ein optimal in Assembler programmierter Parity-Check, der in der hier besprochenen Routine fehlt, gerade noch 38 400 Bit/s beim Empfang ermöglichen. Doch wer braucht schon im Zeitalter der Checksummen und aufwendigen Protokolle oder gar des selbstredundanten Codes, der sich selbst korrigiert, wenn er nur erst einmal, ohne ganze Bytes zu verschlucken, im Puffer angekommen ist, Parity-Checks?

Das Listing ist mit dem Aztec-C-Compiler verfaßt. Mit geringen Änderungen ist es auch mit anderen C-Compiler zu übersetzen. Das Listing finden Sie auch auf der PD-Diskette zum Heft (Seite 114). *rz*

Literaturhinweise:

[1] Commodore Amiga, AMIGA ROM Kernel Reference Manual Devices, Third Edition, Reading 1991


```
 59:                  UBYTE *data,SHORT length);
 60: SHORT  StatSer(register struct IOExtSer *io);
 61: SHORT  ReadSer(register struct IOExtSer *io,char *data,
 62:                 SHORT length,SHORT Timeout);
 63:
 64: /*** Funktionen ***/
 65: /*
 66:  * Start der seriellen Kommunikation. Die Schnittstelle wird
 67:  * mit der gewünschten Baudrate initialisiert (Daten-
 68:  * bit-Anzahl, Stoppbit-Anzahl und Parity). Dabei werden
 69:  * dieselben Bitdefinitionen für Flags verwendet, die in
 70:  * devices/serial.h deklariert sind und auch beim Eröffnen
 71:  * des Amiga-Serial-Devices verwendet werden (siehe Amiga ROM
 72:  * Kernel Reference Manual, Kapitel 13 Seite 383 ff.). Es
 73:  * wird ein Empfangspuffer der Größe BufLen angelegt, in den
 74:  * später zyklisch serielle Daten eingetragen werden. Die
 75:  * Parameter in dieser Funktion repräsentieren alle
 76:  * relevanten Einstellungen.
 77:  */
 78: struct IOExtSer *
 79: OpenSer(ULONG Baud,ULONG BufLen,UBYTE ReadLen,
 80:         UBYTE WriteLen,UBYTE Stop,UBYTE Flags)
 81: {
 82:   register struct IOExtSer *io;
 83:   UBYTE    *SerBuffer=NULL;
 84:   LONG     error;
 85:
 86:   if(!(io=(struct IOExtSer *)AllocMem(
 87:     sizeof(struct IOExtSer) ,MEMF_PUBLIC|MEMF_CLEAR)))
 88:       return(NULL);
 89:
 90:   ParamSer(io,Baud,BufLen,ReadLen,WriteLen,Stop,Flags);
 91:
 92:   if(!(SerInt=(struct Interrupt *)
 93:     AllocMem(sizeof(struct Interrupt),
 94:              MEMF_PUBLIC|MEMF_CLEAR)))
 95:     goto Fail;
 96:
 97:   if(!(SerBuffer=(UBYTE*)AllocMem(io->io_RBufLen,
 98:                          MEMF_PUBLIC|MEMF_CLEAR)))
 99:     goto Fail;
100:
101:   /* Interrupt Struktur initialisieren, damit Exec bei
102:    * Receive-Interrupt zu unserer Routine springt
103:    * Node-Typ ist Interrupt. Klar */
104:   SerInt->is_Node.ln_Type=NT_INTERRUPT;
105:   SerInt->is_Node.ln_Pri=0;
106:   SerInt->is_Node.ln_Name=SERINTNAME; /* Name für Liste */
107:   /* Dieser Pointer wird dem Handler in A1 übergeben */
108:   SerInt->is_Data=(APTR)&RBFData;
109:   SerInt->is_Code=SerialReceiveInt;   /* Interrupt-Handler */
110:   RBFData.InCount=RBFData.OutCount=RBFData.Buffer=SerBuffer;
111:   /* Ringpuffer auf Anfang, gelöscht */
112:   RBFData.io=io;
113:   io->IOSer.io_Device=(struct Device *)&RBFData;
114:
115:   /* alten Interruptvektor merken, damit nach CloseSer wieder
116:    * der normale Handler eingesetzt wird */
117:   PriorInterrupt=SetIntVector(INTB_RBF,SerInt);
118:
119:   /* Seriellen Interrupt erlauben, entsprechendes Bit im
120:    * Interrupt-Enable-Register setzen */
121:   custom.intena=INTF_SETCLR|INTF_RBF;
122:   return(io);
123: Fail:
124:   CloseSer(io);
125:   return(NULL);
126: }
127:
128: /*
129:  * Verändern der Parameter der seriellen Schnittstelle. Diese
130:  * Funktion kann jederzeit aufgerufen werden, um die Baudrate
131:  * oder andere Parameter neu zu setzen. Lediglich die Größe
132:  * des Empfangspuffers ist beizubehalten
133:  */
134: void ParamSer(struct IOExtSer *io,ULONG Baud,
135:               ULONG BufLen,UBYTE ReadLen,UBYTE WriteLen,
136:               UBYTE Stop,UBYTE Flags)
137: {
138:   USHORT b;
139:
140:   io->io_Baud=Baud;
141:   b=BAUDFAKTOR/Baud-1;
142:   if(ReadLen==8 && Flags & SERF_PARTY_ON) b|=0x8000;
143:   custom.serper=b;
144:
145:   io->io_RBufLen=BufLen;
146:   io->io_ReadLen=ReadLen;
147:   io->io_WriteLen=WriteLen;
148:   io->io_StopBits=Stop;
149:   io->io_SerFlags=Flags;
150: }
151:
152: /*
153:  * Freigeben der allokierten Daten, Empfangspuffer etc. und
154:  * Sperren des seriellen Interrupts. CloseSer wird
155:  * aufgerufen, wenn keine serielle Kommunikation mehr
156:  * stattfinden soll (meist beim Programmende).
157:  */
158: void CloseSer(struct IOExtSer *io) {
159:   struct SerialIntData *sid;
160:
161:   /* Sperren des Receive Interrupts */
162:   custom.intena=INTF_RBF;
163:
164:   /* Wiedereinsetzen des AMIGA Interrupt Handlers */
165:   if(PriorInterrupt) {
166:     SetIntVector(INTB_RBF,PriorInterrupt);
167:     PriorInterrupt=NULL;
168:   }
169:
170:   /* Freigeben der Interrupt-Struktur, der IOExtSer-Struktur
171:    * und des Empfangspuffers, wenn sie allokiert wurden */
172:   if(SerInt) {
173:     FreeMem(SerInt,sizeof(struct Interrupt));
174:     SerInt=NULL;
175:   }
176:   if(io) {
177:     if(RBFData.Buffer) {
178:       FreeMem(RBFData.Buffer,io->io_RBufLen);
179:       RBFData.Buffer=NULL;
180:     }
181:     FreeMem(io,sizeof(struct IOExtSer));
182:     io=NULL;
183:   }
184: }
185:
186: /*
187:  * Diese Routine wird vom Hauptprogramm zum seriellen Senden
188:  * einer Anzahl Bytes (z.B. eines Strings) aufgerufen. Die
```

```
189:  * Anzahl steht in length, *data zeigt auf den auszugebenden
190:  * String. Beispiel:
191:  *   WriteSer(io,"Hello World",11);
192:  *   char c='A';
193:  *   WriteSer(io,&c,1);    sendet das Zeichen A
194:  * Bei der Ausgabe jedes Byte wird das Parity-Bit per
195:  * Software berechnet, da die Amiga-Hardware keine
196:  * Möglichkeit der Parity-Bildung vorsieht.
197:  */
198: void WriteSer(register struct IOExtSer *io,
199:               UBYTE *data,SHORT length)
200: {
201:   register USHORT tx,mask,StopBitMask,ParityBit;
202:
203:   if(io->io_StopBits==2)
204:     StopBitMask=0x0003;
205:   else
206:     StopBitMask=0x0001;
207:   StopBitMask<<=io->io_WriteLen;
208:
209:   if(io->io_SerFlags & SERF_PARTY_ON) {
210:     StopBitMask<<=1;
211:     ParityBit=1<<io->io_WriteLen;
212:   }
213:
214:   /* Alle Bytes nacheinander aussenden. Forbid() verwenden
215:    * wir, damit andere Tasks nicht dazwischenfunken und damit
216:    * die Stopp-Bitphase durch Taskwechsel nicht verlängert
217:    * wird */
218:   Forbid();
219:   while(length--) {
220:     tx=*data++;
221:
222:     /* wenn Parity enabled, dann Paritybit berechnen */
223:     if(io->io_SerFlags & SERF_PARTY_ON) {
224:       for(mask=1<<io->io_WriteLen-1;mask!=0;mask>>=1) {
225:         if(tx&mask) tx^=ParityBit;
226:       }
227:       if(io->io_SerFlags & SERF_PARTY_ODD)  tx^=ParityBit;
228:     }
229:
230:     /* Warten, bis der Transmitter-Buffer leer ist, bevor das
231:      * nächste Byte gesendet wird */
232:     while(!(custom.serdatr & TBE));
233:
234:     /* Interrupt-Pending-Bit für Transmitter (Interrupt wird
235:      * nicht benötigt) zurücksetzen */
236:     custom.intreq=INTF_TBE;
237:
238:     /* Byte mitsamt Stopp-Bits und Parity ins serielle
239:      * Datenregister schreiben: Senden läuft */
240:     custom.serdat=tx|StopBitMask;
241:   }
242:   Permit();
243: }
244:
245: /*
246:  * Gibt an, ob und wieviele Byte im Empfangspuffer vorhanden
247:  * sind. Wartet beispielsweise das Hauptprogramm auf 8 Bytes,
248:  * könnte das folgende Sequenz vornehmen:
249:  *   char Daten[8];
250:  *   while(StatSer(io)<8);
251:  *   ReadSer(io,&Daten,8,0);
252:  */
253: SHORT StatSer(struct IOExtSer *io) {
254:   register struct SerialIntData *
255:            rbf=(struct SerialIntData *)(io->IOSer.io_Device);
256:
257:   return(rbf->Inhalt);
258: }
259:
260: /*
261:  * Liest aus dem Empfangspuffer eine bestimmte Anzahl Bytes
262:  * nach *data. Wenn weniger Daten im Puffer sind, kehrt die
263:  * Routine nach Ablauf eines Time-Outs zurück, ansonsten
264:  * sofort nach dem Auslesen der Bytes aus dem Receive-Puffer.
265:  * Da Time-Out mittels einer Programmschleife realisiert ist,
266:  * sollte der Wert bei Amigas mit MC68030-Prozessor
267:  * entsprechend höher gesetzt werden. Besser ist
268:  * selbstverständlich die Nutzung des Timer-Devices, hier
269:  * allerdings aufgrund des zur Verfügung stehenden Platzes
270:  * nicht realisierbar. Das hier angegebene Verfahren dient
271:  * nur Demonstrationszwecken. Zurückgeliefert wird die
272:  * tatsächlich gelesene Anzahl Bytes (io->Actual).
273:  */
274: SHORT ReadSer(struct IOExtSer *io,char *data,
275:               SHORT length,SHORT Timeout)
276: {
277:   register struct SerialIntData *
278:            rbf=(struct SerialIntData *)(io->IOSer.io_Device);
279:   register SHORT i,j,k;
280:
281:   io->IOSer.io_Data=(APTR)data;
282:   io->IOSer.io_Length=length;
283:   io->IOSer.io_Actual=0;
284:
285:   for(i=0;i<Timeout;i++) {
286:     /* diese Schleife soll ca. 1 ms dauern */
287:     if(length==0)
288:       break; /* Hauptprogramm will keine Daten */
289:     if(rbf->Inhalt) {
290:       *data++=*rbf->OutCount;
291:       if(++rbf->OutCount>=rbf->Buffer+io->io_RBufLen)
292:         rbf->OutCount=rbf->Buffer;
293:       io->IOSer.io_Actual++;
294:       length--;
295:       rbf->Inhalt--;
296:     } else {
297:       for(j=0;j<100;j++);
298:     }
299:   }
300:   return(io->IOSer.io_Actual);
301: }
302:
303: /*
304:  * Und jetzt die Hauptsache: Schnelle serielle
305:  * Empfangsroutine in Assembler. Jedes serielle Byte löst den
306:  * Receive-Interrupt aus, so daß diese Routine angesprungen
307:  * wird. Zum besseren Verständnis ist für jede Aktion eine
308:  * Entsprechung in C als Kommentar hinzugefügt.
309:  */
310:
311: /* Receive-Puffer-Full-Driver, Interrupt-Routine */
312:
313: #asm
314:   public  _SerialReceiveInt
315:
316: _SerialReceiveInt:
317: ;register a1 struct SerialIntData *rbf;
318: ;register d4 SHORT rx;
319:
320: ;rx=custom.serdatr;
321:   movem.l d4,-(sp)        ; d4 auf den Stack retten
322:   move.w  $dff018,d4      ; serielles Byte nach d4 einlesen
323:
324: ; custom.intreq=INTF_RBF;
325: ; Receive-Interrupt-Pending-Flag rücksetzen
326:   move.w  #$0800,$dff09c
327:   move.l  (a1),a0         ; prüfen, ob im Ringpuffer noch
328:                           ; Platz ist, sonst Byte verwerfen
329:   move.l  52(a0),d0
330:   move.w  16(a1),d1       ;Inhalt >RBufLen ?
331:   cmp.w   d0,d1
332:   bcc     ..52
333:
334: ; *rbf->InCount=rx;
335:   move.l  4(a1),a0        ; Byte in Ringpuffer eintragen
336:   move.b  d4,(a0)
337:
338: ;  rbf->Inhalt++;
339:   add.w  #1,16(a1)        ; Ringpuffer-Zähler inkrementieren
340:
341: ;  if(++rbf->InCount >= rbf->Buffer+rbf->io->io_RBufLen)
342: ;    rbf->InCount=rbf->Buffer;
343:
344:   add.l  #1,4(a1)         ; Ringpuffer-InCounter erhöhen
345:   move.l  (a1),a0         ; wenn Ende des Puffers erreicht,
346:                           ; dann InCounter
347:   move.l  52(a0),d0       ; wieder auf Anfang des Puffers
348:   add.l  12(a1),d0
349:   move.l  4(a1),a0
350:   cmp.l  d0,a0
351:   bcs  ..50
352:   move.l  12(a1),4(a1)
353:
354: ; if(rx&OVRUN) rbf->io->IOSer.io_Error=SerErr_BufOverFlow;
355: ..50
356:   btst.l  #15,d4          ; Overrun Error im Empfangsbyte ?
357:   beq  ..51
358:
359: ..52
360:   move.l  (a1),a0         ; setze entsprechendes Bit in
361:                           ; IOSer->io_Error
362:   move.b  #12,31(a0)
363:
364: ..51
365:    movem.l (sp)+,d4        ; d4 von Stack holen, Empfangsende
366:    rts
367:
368: #endasm                © 1993 M&T
```

»Fastserial.c«: Die schnelle Routine läßt sich in schon bestehende Programme problemlos implementieren



3-D-Darstellung

Rot, Grün, 3D

Jeder hat sie schon bestaunt: Monitore mit animierten 3-D-Darstellungen sind Anziehungspunkte auf jeder Computermesse. Die dritte Dimension auf dem Bildschirm hat selbst für Computerprofis noch immer den Hauch des Sensationellen. Leider gilt: Je realistischer die Darstellung, desto teurer die Hardware. Dabei genügen für den Hausgebrauch schon zwei Mark für eine Rot/Grün-Filterbrille, wie man sie vom Jahrmarkt aus den 3-D-Kinos kennt. Zusätzlich brauchen Sie noch etwas Geschick bei der Belegung der Farbtabelle Ihres Rechners...

von Bernfried Brüggemann

Zunächst zu den Grundlagen, wie man Objekte per Rot/Grün-Verfahren dreidimensional darstellt: Mit unseren beiden Augen sehen wir wegen des unterschiedlichen Blickwinkels leicht voneinander abweichende Bilder unserer Umgebung. Man kann sich das schnell verdeutlichen, wenn man den Daumen am ausgestreckten Arm betrachtet und abwechselnd das rechte und linke Auge zukneift. Der Daumen scheint vor dem Hintergrund hin- und herzuspringen. Der »Daumensprung« wird immer größer, je mehr man die Hand dem Gesicht nähert.

Wenn man mit beiden Augen schaut, erzeugt das Gehirn aus den unterschiedlichen Bildern den Eindruck der räumlichen Tiefe. Man erkennt unmittelbar, in welcher Entfernung sich ein Gegenstand befindet. Das nutzt das Rot/Grün-Verfahren für die 3-D-Darstellung aus. Es erzeugt für jedes Auge ein separates Bild. Das eine enthält nur grüne, das andere nur rote Farben.

Man kann die beiden Bilder kombinieren und beispielsweise auf einem Bildschirm überlagern. Damit jedes Auge nur das zu seinem Blickwinkel gehörende Teilbild sieht, wird ein Auge mit einem roten, das andere mit einem grünen Filter abgedeckt. Da Rot und Grün Komplementärfarben sind, absorbiert der Grünfilter den Rotanteil des Monitorbilds vollständig, der Rotfilter leistet dasselbe für die grüne Farbe. Durch eine Rot/Grün-Brille gewinnt der Betrachter somit einen räumlichen Eindruck der abgebildeten Gegenstände.

	Monitorbild	durch Rotfilter betrachtet	durch Grünfilter betrachtet
Addition rot + grün			
Addition grün + rot			
korrekte überlagerung von rot & grün			

Bild 1: Addition ist keine Überlagerung; wenn man wie im Bild unten eine rote und grüne Kugel überlagert, müßte die Schnittfläche bei Überlagerung gelb sein

Teuflisches

Als pragmatischer Programmierer denkt man sich: »Rot/Grün-3-D (R/G), das müßte doch eigentlich eine der leichtesten Übungen für den Farbenzauberer Amiga sein«, und geht frisch ans Werk. Um sich nicht mit Vektorrechnung und Projektionsgeometrie zu plagen, wählt man am besten ein einfaches Beispiel und begnügt sich zunächst mit Scheibchen, die dank R/G-Verfahren 3-D-real vor und hinter dem Bildschirm schweben sollen.

Für das rechte Auge nimmt man den roten Filter, für das linke den grünen. Scheibchen, die vor dem Bildschirm liegen, erscheinen dann in der grünen Ansicht nach rechts verschoben, in der roten nach links. Für Scheibchen hinter der Bildschirmebene liegen die Verhältnisse genau umgekehrt. Für Scheibchen in der Bildschirmebene fallen rote und grüne Ansicht zusammen. Jetzt braucht man nur noch mit einigen »AreaCircle()«-Anweisungen die Scheiben auf den Bildschirm zu bringen. Leider begegnet man an dieser Stelle dem Teufel im Detail. Es ist nämlich nicht ohne weiteres möglich, einfach rote und grüne Ansicht nacheinander auf den Schirm zu zeichnen.

Wenn der Scheibchendurchmesser größer ist als der Versatz zwischen den beiden Ansichten gibt es einen Rot/Grün-Überlappungsbereich (siehe Bild 1; links). Dieser Kernbereich muß für beide Augen in der gleichen Helligkeit erscheinen wie der Scheib-

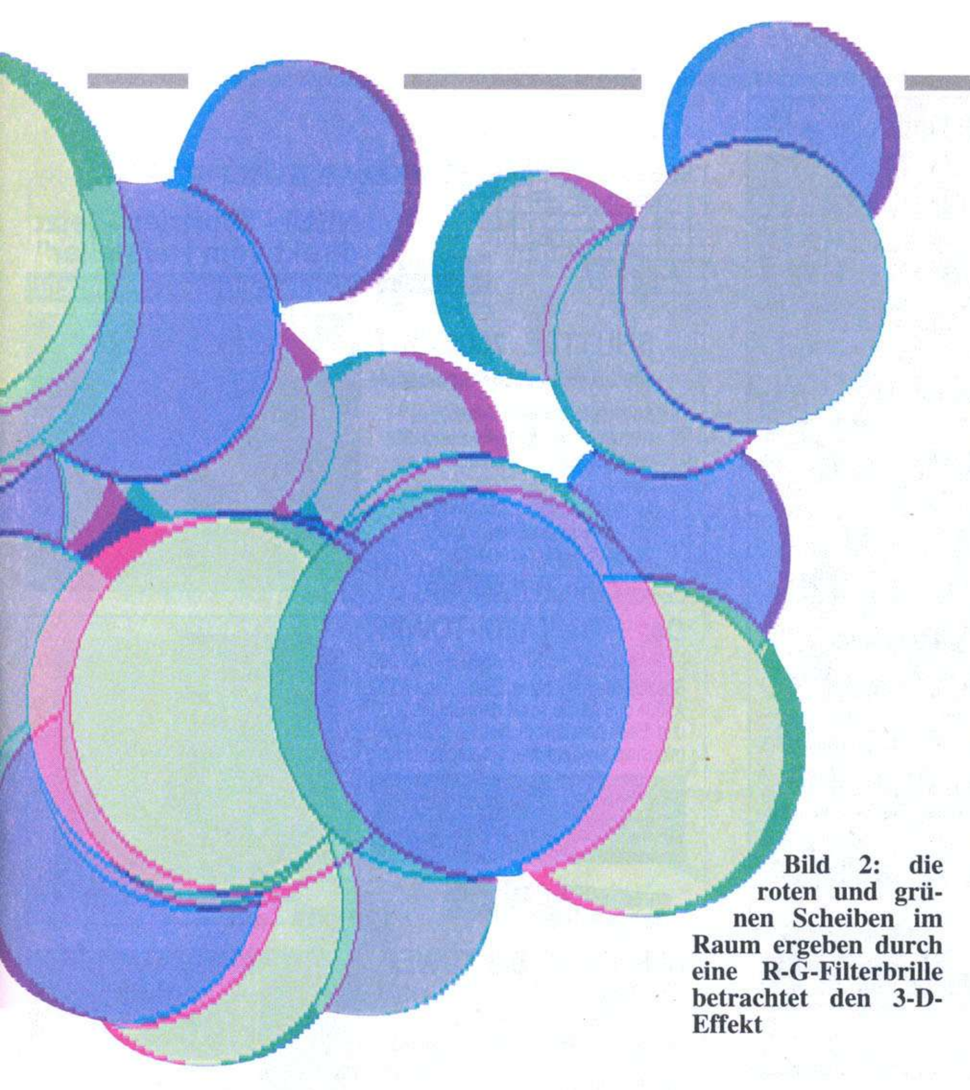


Bild 2: die roten und grünen Scheiben im Raum ergeben durch eine R-G-Filterbrille betrachtet den 3-D-Effekt

den Bitplanes schreibgeschützt, beim Zeichnen der grünen entsprechend die oberen beiden Planes.

Den vier Helligkeitsstufen A bis D werden dabei die Stifte 15, 10, 5 und 0 zugeordnet. Obwohl explizit nur mit vier Stiften gezeichnet wird, entstehen durch die Bit-Maskierung im überlagerten Bild automatisch alle 16 erforderlichen Mischfarben. Wenn man z.B im roten Teilbild eine Scheibe mit Stift 15 zeichnet und diese Scheibe im grünen Teilbild Flächen mit unterschiedlichen Grün-Helligkeitsstufen bedeckt, erscheinen die Flächen durch die Bitmaskierung in der Überlagerung automatisch so, als wären sie mit Stift 12 bis 15 gezeichnet. Anhand der Spalte »Bitbelegung« in der Tabelle kann man sich das leicht verdeutlichen. Wenn die Amiga-Farbtabelle nach den Vorgaben in den Spalten R, G,B modifiziert ist, besitzen alle Mischfarben den richtigen Rot- und Grünanteil und man erhält bei der Überlagerung ein korrektes R/G-3D-Bild. Der Blauanteil muß übrigens nicht unbedingt 0 sein. Er muß nur für alle 16 Mischfarben gleich sein.

Demo-Programm

Das Modula-2-Programm »RGDemo.mod« (Seite 18 bis 20, Programm und Quellcode auf der Diskette zum Heft) bringt das beschriebene Scheibchenarrangement auf den Bildschirm (Bild 2). Um eine Hidden-Line-Darstellung zu erzielen, werden weiter entfernte Scheiben vor den näher liegenden gezeichnet. Die Prozedur »SetWrMsk()«

chenrest. Wenn die rote Ansicht zuletzt gezeichnet wird, ist dieser Kernbereich aber rot und durch das Grünfilter nicht zu sehen. Der Kernbereich müßte gelb gezeichnet werden. Gelb enthält zu gleichen Teilen Rot und Grün. Der Kernbereich erscheint dann für beide Augen in der richtigen Helligkeit und fügt sich mit dem roten und grünen Rand zu einem runden Scheibchen zusammen.

Falls mehrere, sich teilweise verdeckende Scheibchen unterschiedlicher Helligkeit am Bildschirm zu sehen sein sollen, sind eine Vielzahl kompliziert geformter Überlappungsflächen mit verschiedensten R/G-Mischfarbanteilen darzustellen (Bild 2). Auf den ersten Blick scheint dann eine korrekte R/G-Überlagerung nur mit großem Aufwand möglich zu sein . . .

Bits mit Maske

. . . deshalb sollte man noch einen zweiten Blick riskieren. Die Tabelle in Bild 4 auf Seite 19 gibt eine Übersicht der möglichen Farbkombinationen bei vier Helligkeitsstufen im R/G-Bild. Bei der korrekten Überlagerung von roter und grüner Ansicht kann im Prinzip jede der vier roten Helligkeitsstufen mit jeder der vier grünen zusammentreffen. Insgesamt können also 16 Mischfarben im überlagerten Monitorbild auftreten.

Rot- und Grünanteil der Mischfarben entsprechen jeweils den Helligkeitsstufen im roten und grünen Teilbild. Ein Trick macht es möglich, diese Mischfarben auf den Bildschirm zu bringen, ohne daß die Überlappungsflächen explizit mit einem Stift der entsprechenden Farbe gezeichnet werden müßten. Der Trick besteht darin, die für die Darstellung von 16 Mischfarben erforderlichen 4 Bitplanes in zwei Gruppen zu je zwei Bitplanes aufzuteilen. Beim Zeichnen der roten Ansicht werden nun einfach die unteren bei-

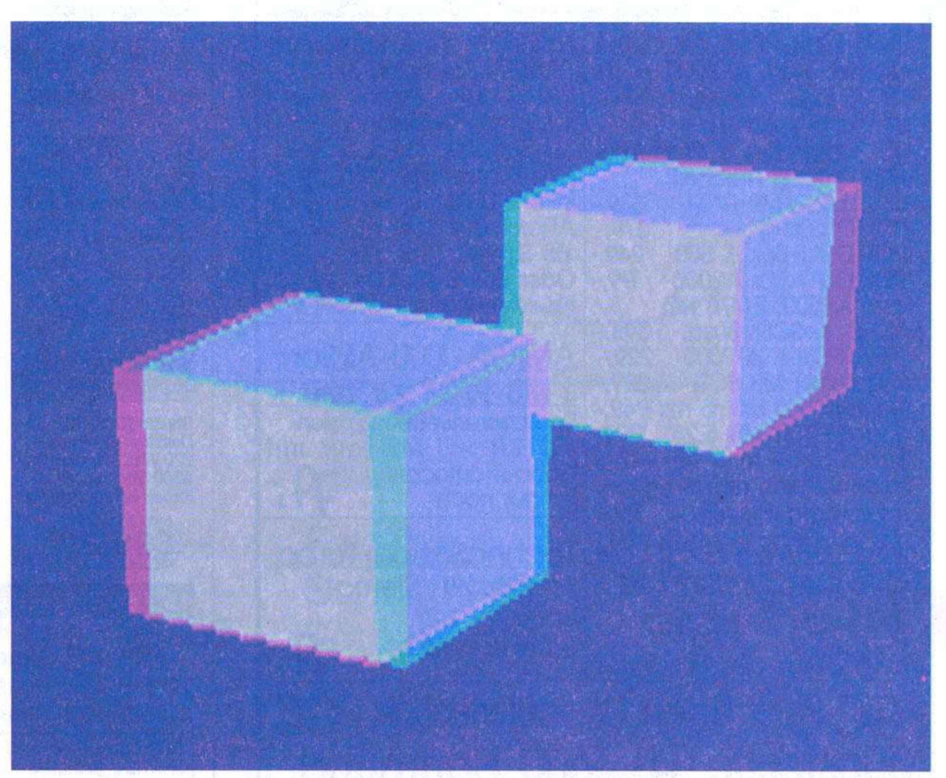


Bild 3: Schnappschuß aus der 3-D-Würfel-Animation. Wie in Bild 2 ist eine R-G-Filterbrille für den 3-D-Eindruck erforderlich.

erledigt die Bitmaskierung bei der R/G-Überlagerung. Als Schmankerl folgt eine 3-D-Animation von zwei um den gemeinsamen Schwerpunkt rotierenden Würfeln. Eigentlich fehlt nur noch das Geräusch zersplitternden Glases, wenn die Würfel den Bildschirm durchschlagen. Um das Listing abdruckbar zu halten, haben wir darauf verzichtet.

Damit der Bewegungsablauf der Animation einigermaßen flüssig bleibt, berechnet der Amiga die umfangreichen Matrizenoperationen im voraus und ruft bei der Darstellung der Szene nur noch die in einem ARRAY gespeicherten Ergebnisse ab. Um die Szene zu berechnen, braucht ein Amiga 500 immerhin ca. zwei Minuten. ub

Zusatzhardware

Rot- und Grünfilter kann man zwar teuer beim Fotografen kaufen, aber billige Klarsichtfolien aus dem Büromaterial-Fachgeschäft genügen vollkommen. Sie sollten nur darauf achten, daß rote und grüne Folie hintereinandergelegt völlig schwarz erscheinen, sonst werden die Teilbilder nicht sauber getrennt. Es ist auch möglich, daß die Farben der Filter nicht ganz genau zu den vom Monitor dargestellten Farben passen. In beiden Fällen kann man die Filterwirkung verbessern, indem man mehrere Folien einer Farbe übereinanderschichtet.

Grenzen: Da beim R/G-Verfahren die räumliche Tiefe durch den Abstand der Teilbildkomponenten erzeugt wird, ist das Tiefenauflösungsvermögen durch den Pixelabstand am Bildschirm begrenzt. Das schränkt den Anwendungsbereich ein. Einen 3D-Solid-Model-Editor wird man nicht befriedigend realisieren können. Eher ist ein Einsatz im Multimedia- oder im Spielebereich denkbar. Das R/G-Verfahren ist z.B. als Option zur Spielbrettdarstellung im Spielprogramm SOGO (Programm des Monats im AMIGA-Magazin 5/92) enthalten. Blockout wäre ein anderes R/G-3D-Projekt. Urteilen Sie selbst! Ist Rot/Grün eine Alternative?

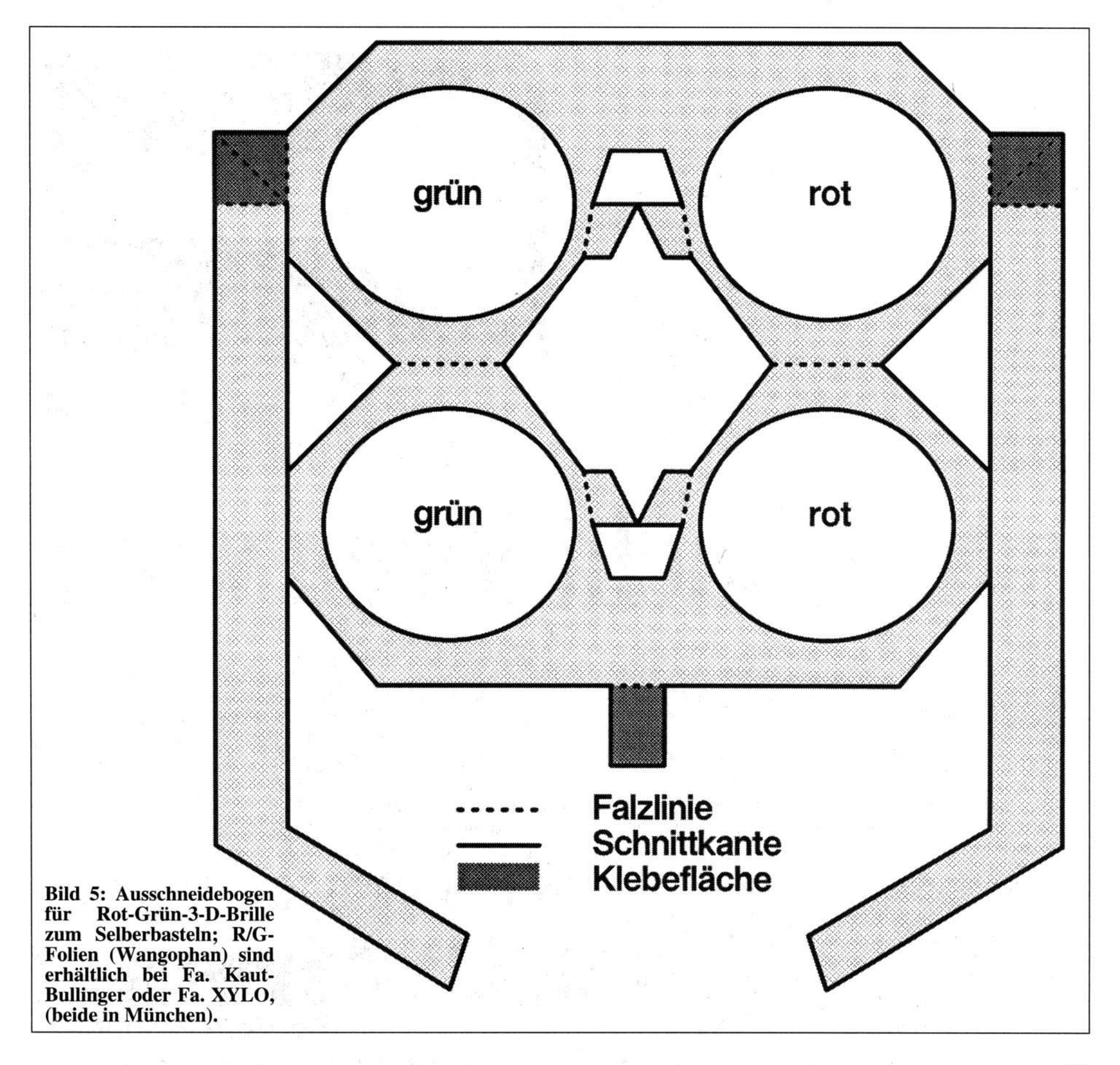


Bild 5: Ausschneidebogen für Rot-Grün-3-D-Brille zum Selberbasteln; R/G-Folien (Wangophan) sind erhältlich bei Fa. Kaut-Bullinger oder Fa. XYLO, (beide in München).

Endlich auch auf deutsch!



Was ist so besonderes an diesem Spiel ?

Wie erklärst du der Welt die Möglichkeiten in deiner Heimatstadt ?

Wie bewältigt ein stetig wachsender Software-Hersteller *all diese Rechnungen* ?

Wie beschäftigst du deine Kinder an einem regnerischen Samstag-Nachmittag ?

CanDo macht Ideen wahr.

Szenario I: Du sitzt an deinem Amiga und fragst dich warum es kein Datenbank/Quizprogramm gibt (das Kinder lieben würden). Die Zeit verrinnt. Frustration setzt ein.

Szenario II: Du hast dir gerade ein hervorragendes Programm ausgedacht - als CanDo-Besitzer erstellst du es selber: Fenster, Fonts, Menus, Animationen, Sound und vieles mehr, alles unter deiner Kontrolle!

Immernoch unsicher?
Dann mach die CanDo-Probefahrt und entdecke, warum bereits tausende von Amiga-Besitzern in den USA täglich mit CanDo arbeiten. Das CanDo-Testdrive kostet nur 29 DM (Demoversion). Und weil wir wissen, daß dir CanDo gefallen wird, nehmen wir dein Testdrive voll in Zahlung!

CanDo bietet Lösungen!
Das besondere an CanDo ist, daß *du* mit minimalem Zeitaufwand, exakt das bekommst, was *du* willst. **Wenn du ein "richtiger" Programmierer bist,** bedenke daß mit CanDo viele große Projekte schnell und einfach erledigt sind.

Los geht's. Probier es aus!
Wir sind sicher, daß du CanDo lieben wirst. CanDo bringt dich schneller ans Ziel.

CanDo V2.0 nur 299 DM!

INOVAtronics

Be More Productive.

Inovatronics GmbH, Im Heidkamp 11, 5000 Köln 91, Tel: 0221-875 126, Fax: 0221-870 47 47

mögliche Kombinationen bei der Überlagerung von roter und grüner Ansicht								Komponenten der Mischfarbe			Bitbelegung				Pen nr.
Helligkeit im roten Teilbild				Helligkeit im grünen Teilbild											
hell	halb hell	halb dunkel	dunkel	hell	halb hell	halb dunkel	dunkel	R %	G %	B %	Bitmap rot		Bitmap grün		
x				x				100	100	0	1	1	1	1	15
x					x			100	66	0	1	1	1	0	14
x						x		100	33	0	1	1	0	1	13
x							x	100	0	0	1	1	0	0	12
	x			x				66	100	0	1	0	1	1	11
	x				x			66	66	0	1	0	1	0	10
	x					x		66	33	0	1	0	0	1	9
	x						x	66	0	0	1	0	0	0	8
		x		x				33	100	0	0	1	1	1	7
		x			x			33	66	0	0	1	1	0	6
		x				x		33	33	0	0	1	0	1	5
		x					x	33	0	0	0	1	0	0	4
			x	x				0	100	0	0	0	1	1	3
			x		x			0	66	0	0	0	1	0	2
			x			x		0	33	0	0	0	0	1	1
			x				x	0	0	0	0	0	0	0	0
15	10	5	0	15	10	5	0								
für die 4 R/G-Helligkeitsstufen explizit verwendete Pen-Nummern															

Bild 4. Die geeignete Belegung der Farbtabelle ergibt zusammen mit der Bit-Maskierung automatisch eine korrekte Überlagerung von rotem und grünem Teilbild

```
 1: MODULE RGDemo;
 2:
 3: FROM SYSTEM     IMPORT ADDRESS,ADR,INLINE,FFP;
 4: FROM Intuition IMPORT NewWindow,IDCMPFlags,IDCMPFlagSet,ScreenPtr,W
    indowPtr,WindowFlags,WindowFlagSet,NewScreen,customScreen,OpenScree
    n,CloseScreen,OpenWindow,CloseWindow,ScreenFlags,ScreenFlagSet,Intu
    iMessagePtr,ScreenToFront;
 5: FROM Graphics  IMPORT AllocRaster,TmpRas,AreaInfo,AreaEllipse,AreaM
    ove,Text,AreaDraw,AreaEnd,BitMap,InitBitMap,ViewModeSet,ViewModes,F
    reeRaster,DrawEllipse,SetRast,Move,InitArea,InitTmpRas,LoadRGB4,Set
    APen,InitRastPort,RastPort,RastPortPtr,Draw;
 6: FROM GfxMacros IMPORT SetWrMsk,SetOPen;
 7: FROM Exec       IMPORT GetMsg,ReplyMsg;
 8: FROM InOut      IMPORT WriteString,WriteLn;
 9: FROM Arts       IMPORT TermProcedure;
10: FROM RandomNumber IMPORT RND,PutSeed;
11: FROM MathLibFFP   IMPORT sin,cos,pi;
12:
13: CONST     rotsteps       = 360; (* Anzahl der Rot.schritte der Würfe
    lanimation *)
14:           p  = 0.2; (* Perspekt. Verkürzungsfaktor *)
15:           alfa  = -70.0*pi/180.0; (* Winkel, unter dem die Rotation
    s-
16:                                              ebene der Würfelanim. ges
    ehen wird  *)
17:    dphi   = pi/180.0; (* Rotationswinkel-Inkrement *)
18:      XShift  = 320; (* Offset, um die Animation in die Bildschirm-
    *)
19:      YShift = 128; (* mitte zu rücken *)
20:      ZFocus = 1000.0; (* z-Koord. d. Fluchtpunktes (0|0|ZFocus) *)
21:
22: TYPE      Ecken  = ARRAY[1..8],[1..3] OF FFP;
23:    FlaechenIndex = ARRAY[1..6] OF INTEGER;
24:           ZvonFCenter = ARRAY[1..6] OF FFP;
25:
26: VAR RGWindow  : NewWindow;
27:     RGScreen     : NewScreen;
28:     RGWindowPtr0 : WindowPtr;
29:     RGScreenPtr0 : ScreenPtr;
30:     RGWindowPtr1 : WindowPtr;
31:     RGScreenPtr1 : ScreenPtr;
32:     RGWinRPPtr0  : RastPortPtr;
33:     RGWinRPPtr1  : RastPortPtr;
34:     RGRPPtr  : RastPortPtr;
35:     Buffer        : ARRAY[0..49] OF CARDINAL;
36:     AreaMemPtr   : ADDRESS;
37:     RGDemoTmpRas : TmpRas;
38:     RGDemoAreaInfo : AreaInfo;
39:     IntuiMsg,IntuiMsg1 : IntuiMessagePtr;
40:     ok,EX,ausserhalb : BOOLEAN;
41:     class  : IDCMPFlagSet;
42:
43:     WP1org  : Ecken; (* Ecken d. 1.Würfels  original      *)
44:     WP1    : Ecken; (*                      transformiert *)
45:     WP2org  : Ecken; (*           2.         original      *)
46:     WP2    : Ecken; (*                      transformiert *)
47:     F   : ARRAY[1..6],[1..5] OF INTEGER; (* Würf.flächn *)
48:     FIndexW1  :  FlaechenIndex; (* Numerierung der Flächen *)
49:     FIndexW2  : FlaechenIndex;
50:     ZFCW1  : ZvonFCenter; (* z-Koordinaten der Würf.flächn *)
51:     ZFCW2  : ZvonFCenter;
52:     Trafo  : ARRAY[1..3],[1..3] OF FFP; (* Rot. matrix *)
53:     ARInt  : ARRAY[1..108*rotsteps] OF INTEGER; (* Animat.*)
54:     I1    : LONGINT; (* Index für ARInt *)
55:     RGDelta  : FFP;     (* Faktor für Rot/Grün-Versatz *)
56:
57: PROCEDURE CloseDown ; (* räumt zum Schluß alles auf *)
58: VAR i,j  : INTEGER;
59: BEGIN (* CloseDown *)
60:       IF RGWindowPtr0 # NIL THEN CloseWindow(RGWindowPtr0); END;
61:       IF RGScreenPtr0 # NIL THEN CloseScreen(RGScreenPtr0); END;
62:       IF RGWindowPtr1 # NIL THEN CloseWindow(RGWindowPtr1); END;
63:       IF RGScreenPtr1 # NIL THEN CloseScreen(RGScreenPtr1); END;
64:       IF AreaMemPtr # NIL THEN   FreeRaster(AreaMemPtr,640,256); EN
    D;
65: END CloseDown ;
```

```
66:
67: PROCEDURE Okay(text : ARRAY OF CHAR ; adr : ADDRESS):BOOLEAN ;
68: BEGIN (* Okay *)
69:   IF adr = NIL THEN
70:       WriteString(text);WriteString(" läßt sich nicht oeffnen !!") ;
   WriteLn ;
71:       WriteString(" ProgrammABBRUCH") ; WriteLn ; RETURN(FALSE);
72:   ELSE RETURN(TRUE); END (* IF *) ;
73: END Okay ;
74:
75: PROCEDURE InitOberfl; (* initialisiert 2 Screens etc. für DoubleBuf
   fering *)
76: VAR i,j :INTEGER;
77: BEGIN
78:    RGScreenPtr0:=NIL; RGWindowPtr0:=NIL;
79:    RGScreenPtr1:=NIL; RGWindowPtr1:=NIL; AreaMemPtr:=NIL;
80:
81:    WITH RGScreen DO
82:       leftEdge := 0 ; topEdge := 0 ; width := 640 ; height := 256 ;
   depth := 4 ;
83:       detailPen := 0 ; blockPen := 1 ; viewModes := ViewModeSet{hir
   es} ;
84:       type := customScreen; font := NIL ; defaultTitle :=ADR("NIL")
   ;
85:       gadgets := NIL ; customBitMap := NIL;
86:    END (* WITH *) ;
87:
88:    RGScreenPtr0 := OpenScreen(RGScreen) ;
89:    IF NOT Okay("RGScreen",RGScreenPtr0) THEN CloseDown;HALT;END;
90:
91:    WITH RGWindow DO
92:     leftEdge:=0; topEdge:=1; width:=640; height:=255;
93:     detailPen:=9; blockPen:=15; idcmpFlags:=IDCMPFlagSet{closeWindo
   w};
94:     flags:=WindowFlagSet{windowClose,gimmeZeroZero,activate};
95:     firstGadget:=NIL; checkMark:=NIL; title:=ADR("Rot-Grün-3D-Demo
   0");
96:     bitMap:=NIL; type:=customScreen; screen:= RGScreenPtr0; minWidt
   h:=600;
97:     maxWidth:=640; minHeight:=256; maxHeight:=256;
98:    END;
99:
100:   RGWindowPtr0:=OpenWindow(RGWindow);
101:   IF NOT Okay("RGWindow",RGWindowPtr0) THEN CloseDown;HALT;END;
102:   RGWinRPPtr0:=RGWindowPtr0^.rPort;
103:
104:   RGScreenPtr1 := OpenScreen(RGScreen) ;
105:   IF NOT Okay("RGScreen",RGScreenPtr1) THEN CloseDown;HALT;END;
106:
107:   WITH RGWindow DO
108:     title:=ADR("Rot-Grün-3D-Demo 1"); screen:= RGScreenPtr1;
109:   END;
110:
111:   RGWindowPtr1:=OpenWindow(RGWindow);
112:   IF NOT Okay("RGWindow",RGWindowPtr1) THEN CloseDown;HALT;END;
113:   RGWinRPPtr1:=RGWindowPtr1^.rPort;
114:
115:   RGRPPtr:=RGWinRPPtr0;
116:
117:   AreaMemPtr:=AllocRaster(640,256);
118:   IF NOT Okay("AreaMem",AreaMemPtr) THEN CloseDown; HALT;END;
119:   FOR i:=0 TO 49 DO Buffer[i]:=0;END;
120:   InitArea(RGDemoAreaInfo,ADR(Buffer),20);   (* AreaInfo initialisi
   eren *)
121:   InitTmpRas(RGDemoTmpRas,AreaMemPtr,20480); (* TmpRas initialisier
   en *)
122:   RGWinRPPtr0^.tmpRas:=ADR(RGDemoTmpRas);    (* TmpRas übergeben *)
123:   RGWinRPPtr0^.areaInfo:=ADR(RGDemoAreaInfo);(* AreaInfo übergeben
   *)
124:   RGWinRPPtr1^.tmpRas:=ADR(RGDemoTmpRas);    (* TmpRas übergeben *)
125:   RGWinRPPtr1^.areaInfo:=ADR(RGDemoAreaInfo);(* AreaInfo übergeben
   *)
126:
127:   PutSeed(2);
128: END  InitOberfl;
129:
130: PROCEDURE FarbTabelle;(* $E- *)
131: BEGIN
132:    INLINE(0007H,0057H,0077H,0097H,
133:           0507H,0557H,0577H,0597H,
134:           0707H,0757H,0777H,0797H,
135:           0907H,0957H,0977H,0997H);
136: END FarbTabelle;
137:
138: PROCEDURE FarbTabelleLaden;
139: BEGIN
140:  LoadRGB4(ADR(RGWindowPtr0^.wScreen^.viewPort),ADR(FarbTabelle),16)
   ;
141:  LoadRGB4(ADR(RGWindowPtr1^.wScreen^.viewPort),ADR(FarbTabelle),16)
142: END FarbTabelleLaden;
143:
144: PROCEDURE ScheibenZeichnen;
145: VAR i,k,r,x,y,deltarot,deltagruen,max : INTEGER;
146:     Pen     : CARDINAL;
147: BEGIN
148:   max:=20;
149:   FOR i:=1 TO 2*max DO
150:      ausserhalb:=TRUE;
151:      WHILE ausserhalb DO
152:         x:=1+RND(640);y:=1+RND(256);
153:         IF i<max THEN deltarot:=max-i;deltagruen:=0;k:=max-i;r:=20+
   i DIV 2;
154:         ELSE           deltarot:=0;deltagruen:=i-max;k:=i-max;r:=20+
   i DIV 2;
155:         END;
156:         IF  (2+2*r+k<x) AND (638-2*r>x)
157:         AND (2+r    <y) AND (240-r    >y) THEN
158:            ausserhalb:=FALSE;
159:         END;
160:      END;
161:      Pen:=5*CARDINAL(RND(3)+1);
162:
163:      FOR k:=0 TO 49 DO Buffer[k]:=0;END;
164: (* rote Ansicht zeichnen *)
165:      SetAPen(RGWinRPPtr1,Pen); SetWrMsk(RGWinRPPtr1,0F3H);
166:      ok:=AreaEllipse(RGWinRPPtr1,x-deltarot,y,2*r,r);
167:      ok:=AreaEnd(RGWinRPPtr1);
168:      SetAPen(RGWinRPPtr1,0);
169:      DrawEllipse(RGWinRPPtr1,x-deltarot,y,2*r,r);
170: (* grüne Ansicht zeichnen *)
171:      SetAPen(RGWinRPPtr1,Pen);SetWrMsk(RGWinRPPtr1,0FCH);
172:      FOR k:=0 TO 49 DO Buffer[k]:=0;END;
173:      ok:=AreaEllipse(RGWinRPPtr1,x-deltagruen,y,2*r,r);
174:      ok:=AreaEnd(RGWinRPPtr1);
175:      SetAPen(RGWinRPPtr1,0);
176:      DrawEllipse(RGWinRPPtr1,x-deltagruen,y,2*r,r)
177:   END;
178:   SetWrMsk(RGWinRPPtr1,0FFH);SetAPen(RGWinRPPtr1,15);
179:   Move(RGWinRPPtr1,20,10);Text(RGWinRPPtr1,ADR
180:  ("AMIGA berechnet jetzt die Würfelanimation, bitte ca. 2 min. warte
   n..."),69);
181: END ScheibenZeichnen;
182:
183: PROCEDURE AnimWuerfel;
184: VAR i,j,k,n  : INTEGER;
185:     m   : LONGINT;
186:
187: PROCEDURE QuickSort(l,r:CARDINAL;VAR Wt:ZvonFCenter;VAR FI:Flaechen
   Index);
188: (* sortiert für die Hiddenline Darstellung die Flächen nach der z-K
   oordinate
189:    des Flächenzentrums *)
190: VAR i,j  : CARDINAL;
191:     Ind  : INTEGER;
192:     x,y  : FFP;
193: BEGIN
194:  i:=l;j:=r; x:=Wt[(l+r) DIV 2];
195:  REPEAT
196:    WHILE Wt[i]<x DO INC(i) END; WHILE x<Wt[j] DO DEC(j) END;
197:    IF i<=j THEN
198:       y:=Wt[i];      Ind:=FI[i];
199:       Wt[i]:=Wt[j]; FI[i]:=FI[j];
200:       Wt[j]:=y;     FI[j]:=Ind;
201:       INC(i);      DEC(j);
202:    END;
203:  UNTIL i>j;
204:  IF l<j THEN QuickSort(l,j,Wt,FI) END;
205:  IF l<r THEN QuickSort(i,r,Wt,FI) END;
206: END QuickSort;
207:
208: PROCEDURE RechneARInt(VAR WP:Ecken;VAR P1,P2,P3,P4,Pen:INTEGER);
209: (* Berechnet die Animationssequenz im voraus und speichert alles in
   ARInt *)
210: VAR j :INTEGER;
211: BEGIN
212:   ARInt[I1]:=Pen;INC(I1);
213:   ARInt[I1]:=INTEGER(WP[P1][1]*(ZFocus-WP[P1][3])/ZFocus
214:                 +RGDelta*WP[P1][3]+0.5)+XShift; INC(I1);
215:   ARInt[I1]:=INTEGER(WP[P1][2]*(ZFocus-WP[P1][3])/ZFocus) DIV 2 +YS
   hift;INC(I1);
216:   ARInt[I1]:=INTEGER(WP[P2][1]*(ZFocus-WP[P2][3])/ZFocus
217:                 +RGDelta*WP[P2][3]+0.5)+XShift; INC(I1);
218:   ARInt[I1]:=INTEGER(WP[P2][2]*(ZFocus-WP[P2][3])/ZFocus) DIV 2 +YS
   hift;INC(I1);
219:   ARInt[I1]:=INTEGER(WP[P3][1]*(ZFocus-WP[P3][3])/ZFocus
220:                 +RGDelta*WP[P3][3]+0.5)+XShift; INC(I1);
221:   ARInt[I1]:=INTEGER(WP[P3][2]*(ZFocus-WP[P3][3])/ZFocus) DIV 2 +YS
   hift;INC(I1);
222:   ARInt[I1]:=INTEGER(WP[P4][1]*(ZFocus-WP[P4][3])/ZFocus
223:                 +RGDelta*WP[P4][3]+0.5)+XShift; INC(I1);
224:   ARInt[I1]:=INTEGER(WP[P4][2]*(ZFocus-WP[P4][3])/ZFocus) DIV 2 +YS
   hift;
225:   IF I1<108*rotsteps  THEN INC(I1)
226:                       ELSE  I1:=1
227:   END;
228: END RechneARInt;
229:
230: BEGIN
231:   I1:=1; RGDelta:=0.04;
232:
233: (* Definition der Würfelflächen anhand der aufspannenden Ecken *)
234:   F[1][1]:=1; F[1][2]:=2; F[1][3]:=3; F[1][4]:=4; F[1][5]:=5;
```

```
235:   F[2][1]:=1; F[2][2]:=2; F[2][3]:=6; F[2][4]:=5; F[2][5]:=5;
236:   F[3][1]:=2; F[3][2]:=3; F[3][3]:=7; F[3][4]:=6; F[3][5]:=10;
237:   F[4][1]:=1; F[4][2]:=4; F[4][3]:=8; F[4][4]:=5; F[4][5]:=5;
238:   F[5][1]:=3; F[5][2]:=4; F[5][3]:=8; F[5][4]:=7; F[5][5]:=5;
239:   F[6][1]:=5; F[6][2]:=6; F[6][3]:=7; F[6][4]:=8; F[6][5]:=10;
240:
241: (* Koordinaten der Würfelecken im KO-System 2;
242:    KO-System 1: Ursprung: linke obere Bildschirmecke;
243:                 x-Achse: oberer Bildschirmrand links -> rechts;
244:                 y-Achse: linker Bildschirmrand oben -> unten;
245:                 z-Achse: in Blickrichtung in den Bildschirm hinein
246: KO-System 2: geht aus KO-System 1 durch Drehung um alfa um die x-Ac
    hse hervor *)
247:
248:   WP1org[1,1]:=-1.5; WP1org[1,2]:= 1.0; WP1org[1,3]:=-1.0;
249:   WP1org[2,1]:=-3.5; WP1org[2,2]:= 1.0; WP1org[2,3]:=-1.0;
250:   WP1org[3,1]:=-3.5; WP1org[3,2]:=-1.0; WP1org[3,3]:=-1.0;
251:   WP1org[4,1]:=-1.5; WP1org[4,2]:=-1.0; WP1org[4,3]:=-1.0;
252:   WP1org[5,1]:=-1.5; WP1org[5,2]:= 1.0; WP1org[5,3]:= 1.0;
253:   WP1org[6,1]:=-3.5; WP1org[6,2]:= 1.0; WP1org[6,3]:= 1.0;
254:   WP1org[7,1]:=-3.5; WP1org[7,2]:=-1.0; WP1org[7,3]:= 1.0;
255:   WP1org[8,1]:=-1.5; WP1org[8,2]:=-1.0; WP1org[8,3]:= 1.0;
256:   FOR i:=1 TO 8 DO
257:       WP2org[i][1]:=WP1org[i][1]+5.0;
258:       WP2org[i][2]:=WP1org[i][2]; WP2org[i][3]:=WP1org[i][3];
259:   END;
260:
261: (* Vergrößerungsfaktor anbringen *)
262:   FOR i:=1 TO 8 DO  FOR j:=1 TO 3 DO
263:       WP1org[i,j]:=WP1org[i,j]*45.0; WP2org[i,j]:=WP2org[i,j]*45.0;
264:   END; END;
265:
266: (* Koordinaten in KO-System 1 umrechnen *)
267:   Trafo[1,1]:= 1.0; Trafo[1,2]:= 0.0;        Trafo[1,3]:= 0.0;
268:   Trafo[2,1]:= 0.0; Trafo[2,2]:= cos(alfa); Trafo[2,3]:=-sin(alfa);
269:   Trafo[3,1]:= 0.0; Trafo[3,2]:= sin(alfa); Trafo[3,3]:= cos(alfa);
270:   FOR i:=1 TO 8 DO FOR j:=1 TO 3 DO
271:       WP1[i,j]:=0.0; WP2[i,j]:=0.0;
272:   END; END;
273:   FOR i:=1 TO 8 DO FOR j:=1 TO 3 DO FOR k:=1 TO 3 DO
274:       WP1[i,j]:=WP1[i,j]+Trafo[j,k]*WP1org[i,k];
275:       WP2[i,j]:=WP2[i,j]+Trafo[j,k]*WP2org[i,k];
276:   END; END; END;
277:   FOR i:=1 TO 8 DO FOR j:=1 TO 3 DO
278:       WP1org[i,j]:=WP1[i,j]; WP2org[i,j]:=WP2[i,j];
279:   END; END;
280:
281: (* Matrix für die Rotation der Würfel um dphi um die z-Achse von KO
    -System 2 *)
282:   Trafo[1,1]:= cos(dphi);
283:         Trafo[1,2]:= cos(alfa)*sin(dphi);
284:               Trafo[1,3]:= sin(dphi)*sin(alfa);
285:   Trafo[2,1]:= -cos(alfa)*sin(dphi);
286:         Trafo[2,2]:= cos(alfa)*cos(alfa)*cos(dphi)+sin(alfa)*sin(alf
    a);
287:               Trafo[2,3]:= cos(alfa)*cos(dphi)*sin(alfa)-sin(alfa)*co
    s(alfa);
288:   Trafo[3,1]:=-sin(alfa)*sin(dphi);
289:         Trafo[3,2]:= sin(alfa)*cos(alfa)*cos(dphi)-cos(alfa)*sin(alf
    a);
290:               Trafo[3,3]:= sin(alfa)*sin(alfa)*cos(dphi)+cos(alfa)*co
    s(alfa);
291:
292:   FOR m:=1 TO rotsteps DO
293:
294: (* neue Koordinaten nach der Rotation um dphi berechnen *)
295:     FOR i:=1 TO 8 DO FOR j:=1 TO 3 DO
296:         WP1[i,j]:=0.0; WP2[i,j]:=0.0;
297:     END; END;
298:     FOR i:=1 TO 8 DO FOR j:=1 TO 3 DO FOR k:=1 TO 3 DO
299:        WP1[i,j]:=WP1[i,j]+Trafo[j,k]*WP1org[i,k];
300:        WP2[i,j]:=WP2[i,j]+Trafo[j,k]*WP2org[i,k];
301:     END; END; END;
302:     FOR i:=1 TO 8 DO FOR j:=1 TO 3 DO
303:         WP1org[i,j]:=WP1[i,j]; WP2org[i,j]:=WP2[i,j];
304:     END; END;
305:
306: (* Abstand der Flächenzentren vom Betrachter berechnen *)
307:     FOR i:=1 TO 6 DO
308:         j:=F[i,1]; k:=F[i,3];
309:         ZFCW1[i]:=(WP1[j,1]+WP1[k,1])*(WP1[j,1]+WP1[k,1])+
310:                   (WP1[j,2]+WP1[k,2])*(WP1[j,2]+WP1[k,2])+
311:                   (WP1[j,3]+WP1[k,3]+50.0*ZFocus)*
312:                   (WP1[j,3]+WP1[k,3]+50.0*ZFocus);
313:         ZFCW2[i]:=(WP2[j,1]+WP2[k,1])*(WP2[j,1]+WP2[k,1])+
314:                   (WP2[j,2]+WP2[k,2])*(WP2[j,2]+WP2[k,2])+
315:                   (WP2[j,3]+WP2[k,3]+50.0*ZFocus)*
316:                   (WP2[j,3]+WP2[k,3]+50.0*ZFocus);
317:     END;
318:     FOR i:=1 TO 6 DO FIndexW1[i]:=i; FIndexW2[i]:=i; END;
319:
320: (* Flächen für Hiddenline-Darstellung sortieren *)
321:     QuickSort(1,6,ZFCW1,FIndexW1); QuickSort(1,6,ZFCW2,FIndexW2);
322:
323:     n:=1;
324:     RGDelta:=-RGDelta;
325:
326: (* für beide Würfel die sichtbaren drei Flächen im Anim-speicher ab
    legen *)
327:     LOOP
328:       IF ZFCW2[1]<ZFCW1[1] THEN
329:          FOR i:=3 TO 1 BY -1 DO
330:              j:=FIndexW1[i];
331:     RechneARInt(WP1org,F[j,1],F[j,2],F[j,3],F[j,4],F[j,5]);
332:   END;
333:   FOR i:=3 TO 1 BY -1 DO
334:              j:=FIndexW2[i];
335:     RechneARInt(WP2org,F[j,1],F[j,2],F[j,3],F[j,4],F[j,5]);
336:   END;
337:       ELSE
338:          FOR i:=3 TO 1 BY -1 DO
339:              j:=FIndexW2[i];
340:       RechneARInt(WP2org,F[j,1],F[j,2],F[j,3],F[j,4],F[j,5]);
341:   END;
342:   FOR i:=3 TO 1 BY -1 DO
343:              j:=FIndexW1[i];
344:     RechneARInt(WP1org,F[j,1],F[j,2],F[j,3],F[j,4],F[j,5]);
345:   END;
346:       END;
347:
348:       IF n=2 THEN
349:           EXIT
350:       ELSE
351:           INC(n); RGDelta:=-RGDelta;
352:       END;
353:     END;
354:   END;
355:
356:   n:=1;
357:   SetWrMsk(RGRPPtr,0FFH); SetRast(RGRPPtr,0);
358:   m:=-8;
359:
360: (* Animation zeichnen *)
361:   LOOP
362:     INC(m,9);
363:     IF m+8>108*rotsteps THEN m:=1 END;
364:     IF (n=1)  THEN SetWrMsk(RGRPPtr,0F3H);SetOPen(RGRPPtr,10); END
    ;
365:     IF (n=7)  THEN SetWrMsk(RGRPPtr,0FCH);SetOPen(RGRPPtr,10); END
    ;
366:     FOR j:=0 TO 49 DO Buffer[j]:=0;END;
367:     SetAPen(RGRPPtr,ARInt[m]);
368:     ok:=AreaMove(RGRPPtr,ARInt[m+1],ARInt[m+2]);
369:     ok:=AreaDraw(RGRPPtr,ARInt[m+3],ARInt[m+4]);
370:     ok:=AreaDraw(RGRPPtr,ARInt[m+5],ARInt[m+6]);
371:     ok:=AreaDraw(RGRPPtr,ARInt[m+7],ARInt[m+8]);
372:     ok:=AreaEnd(RGRPPtr);
373:     IF (n=12) THEN
374:       n:=0;
375:       IF (RGRPPtr=RGWinRPPtr1) THEN
376:          RGRPPtr:=RGWinRPPtr0;ScreenToFront(RGScreenPtr1);
377:       ELSE
378:          RGRPPtr:=RGWinRPPtr1;ScreenToFront(RGScreenPtr0);
379:       END;
380:       SetWrMsk(RGRPPtr,0FFH); SetRast(RGRPPtr,0);
381:     END;
382:     INC(n);
383:
384:     EX:=FALSE;
385:     IF (RGWindowPtr0 # NIL) THEN
386:        IntuiMsg:=GetMsg(RGWindowPtr0^.userPort);
387:        WHILE IntuiMsg#NIL DO
388:            class:=IntuiMsg^.class; ReplyMsg(IntuiMsg);
389:            IF (closeWindow IN class) THEN EX:=TRUE;END;
390:            IntuiMsg:=GetMsg(RGWindowPtr0^.userPort);
391:        END;
392:     END;
393:     IF (RGWindowPtr1 # NIL) THEN
394:        IntuiMsg:=GetMsg(RGWindowPtr1^.userPort);
395:        WHILE IntuiMsg#NIL DO
396:            class:=IntuiMsg^.class; ReplyMsg(IntuiMsg);
397:            IF (closeWindow IN class) THEN EX:=TRUE;END;
398:            IntuiMsg:=GetMsg(RGWindowPtr1^.userPort);
399:        END;
400:     END;
401:     IF EX THEN EXIT;END;
402:   END;
403: END AnimWuerfel;
404:
405: BEGIN
406:   WriteString("Rot-Grün-3D-Demo Version 1.0,");WriteLn;
407:   WriteString("Copyright: Bernfried Brüggemann, Munich 22.04.92");W
   riteLn;
408:   TermProcedure(CloseDown);
409:   InitOberfl;
410:   FarbTabelleLaden;
411:   ScheibenZeichnen;
412:   AnimWuerfel;
413: END RGDemo.
```


»RGB_Demo.mod«: Das Modula-2-Programm (M2Amiga) stellt Objekte per Rot-Grün-Verfahren dreidimensional dar

Ein praxisorientiertes Verfahren

Zwei- und dreidimensionales Clipping

Wichtigste Aufgabe von Programmen ist es heute, den Bildschirm zu beschreiben, zu färben, zu zeichnen oder zu scrollen. Das Verhältnis zu den wenigen, die Motoren steuern, Meßwerte aufnehmen oder Impulse ausgeben, dürfte ca. 9:1 sein. Grund genug, über effiziente Grafikverfahren nachzudenken. Hier finden Sie leistungsfähige Algorithmen vor.

von Bernhard Emese

Der Bildschirm ist des Computers liebstes Kind, und die Kunst des Erzeugens von Linien und Buchstaben ist gleichermaßen wichtig wie der wissenschaftliche Dialog oder das Abballern feindlicher Raumschiffe. Doch auch hier werden zur Beherrschung manche Kniffe verlangt. Einer davon ist das »Clipping«. Darunter versteht man das rechtzeitige Abschneiden (Kappen) von Linien, falls diese über den Bildrand hinauslaufen sollen.

Gerade bei 3-D-Animationen, wo mit Hilfe der Maus jeder beliebige Blickwinkel einzustellen ist und das Objekt so vollständig aus dem Bild hinauslaufen kann, ist Clipping also unvermeidlich. Setzt man Clipping nicht ein, würden die Linien, die über den rechten Rand hinauslaufen, um eine Zeile nach unten versetzt am linken Rand wieder auftauchen. Verläßt darüber hinaus eine Linie am oberen oder unteren Rand den sichtbaren Bereich, schreibt man so Bilddaten außerhalb des Video-RAMs in verbotene Adressen, was üblicherweise zum Absturz führt.

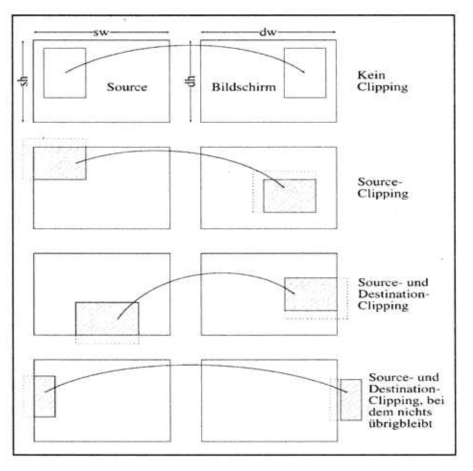


Bild 1: Das prinzipielle Vorgehen unserer Funktion ClpBltBitMap – so überprüft sie die Dimensionen des zu kopierenden Bereichs

Es genügt aber nicht, Linien und andere Bildelemente, die nur teilweise zu sehen sind, vollständig wegzulassen. Das Clippen zerlegt ein Objekt in sichtbare und unsichtbare Elemente und sorgt dafür, daß nur die sichtbaren gezeichnet werden.

Doch wofür eigentlich noch 2-D-Clipping, schließlich ist es doch schon Bestandteil des Amiga-Betriebssystems, repräsentiert durch die »layers.library«? Jeder Aufruf der OpenWindow()-Funktion liefert uns einen Zeiger auf den RastPort unseres Fensters. Über den RastPort ist es dann möglich, auf den verantwortlichen Layer zuzugreifen. Dieser Layer veranlaßt die »graphics.library«, jeden Zeichenbefehl automatisch und falls notwendig zu clippen. Ersetzt man den Layer-Pointer probeweise durch null und zeichnet anschließend eine Linie, stellt man fest, daß es jetzt möglich ist, über die Fenstergrenzen hinaus zu zeichnen. Selbstverständlich ist der Layer-Pointer wieder auf den ursprünglichen Wert zu setzen, bevor wir das Fenster schließen – sonst bleiben Speicherleichen übrig.

Bis heute (Betriebssystem OS-2.0) ist der Clipping-Algorithmus noch immer nicht fehlerfrei. Bei großen X- bzw. Y-Werten (z.B. Draw(RPort,50000,100000)) versagt er und verursacht Systemabstürze. 3-D-Clipping hingegen ist Betriebssystemfremd und ist insofern sowieso zu implementieren.

Doch bleiben wir zunächst im Zweidimensionalen. Unsere Aufgabe ist es, das Zeichnen nur in einem definierten rechteckigen Bereich zuzulassen und alle Pixel (Bildpunkte), die diesen verlassen, nicht abzubilden. Clipping innerhalb beliebiger Umrisse macht man im Amiga am besten über Masken-Bitplanes, die u.a. bei Zeichenoperationen des Blitters Verwendung finden.

2-D-Clipping und die Begrenzung des Bildschirmfensters

Wir unterscheiden im 2-D-Raum zwischen zwei oft anzutreffenden Clipping-Verfahren:

❒ Clippen beim Plazieren von rechteckigen Bildteilen, z.B. mit den Funktionen RectFill(), BltBitMap() oder Text().

❒ Clippen beim Zeichnen von Linien mit z.B. Move() oder Draw().

In beiden Fällen soll das Clipping mit möglichst wenigen Befehlen, also optimiert, erfolgen. Bei einem dreidimensionalen Ballerspiel müssen schließlich Zehntausende solcher Linien gezeichnet werden. Durchdachte Algorithmen sind hier entscheidend.

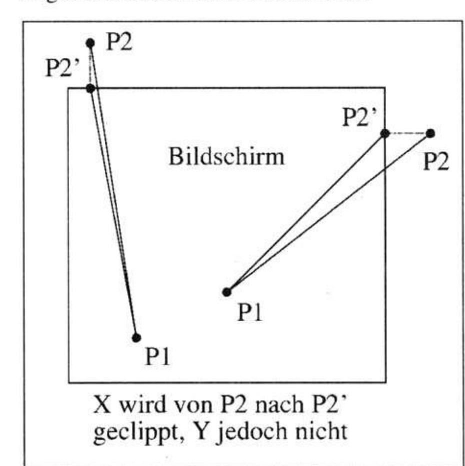


Bild 2: Dieses Vorgehen führt zum falschen Ergebnis, da nur eine Koordinate berücksichtigt bzw. verkürzt wird – beide Punkte sind zu beachten

Das klassische Verfahren für rechteckige Bereiche funktioniert über die IF-Abfrage. Hier sind hintereinander mehrere Bedingungen zu checken, die feststellen, wo abgeschnitten wird und wieviel darzustellen ist.

In der graphics.library finden wir den Befehl ClipBlit(). Der Name verrät, wo diese Funktion eingesetzt wird: beim Kopieren eines rechteckigen Bildausschnitts von einem RastPort zum anderen inkl. Clipping. Der Nachteil dieser Funktion ist, daß das Zeichnen mit dem Elektronenstrahl synchronisiert

wird. Das vermeidet zwar einerseits unschönes Flackern, andererseits geht alles sehr langsam. Für Animation jedenfalls ist es völlig indiskutabel, und das Flackern bereitet uns beim Double-Buffering sowieso keine Kopfschmerzen.

Doch es gibt noch eine andere Funktion: BltBitMap(). An Geschwindigkeit ist diese kaum zu übertreffen, zeichnet aber gnadenlos über alle Bildschirmbegrenzungen hinaus. Die im Listing vorgestellte Routine verbindet beides miteinander, indem sie die Koordinaten vor dem BltBitMap-Aufruf clippt.

Der Funktion ClipBltBitMap übergeben wir neben den BltBitMap-Parametern zusätzlich sw (Sourcewidth, die Breite des Originalausschnitts), sh (Sourceheight, die Höhe des Originalausschnitts), dw (Displaywidth) und dh (Displayheight). Sie repräsentieren die Clip-Grenzen. Ansonsten ist sie mit der des BltBitMap-Kommandos identisch.

Zunächst überprüfen wir, ob sx, sy, sx+sizex und sy+sizey im erlaubten Bereich liegen – man bezeichnet das als Source-Clipping. Ist das nicht der Fall, stutzen wir sie zunächst zurecht. Danach führen wir das Ziel-Clipping aus: dx, dy, dx+sizex und dy+sizey checken wir wiederum und verkleinern eventuell den Zielbereich. Anschließend darf gezeichnet werden.

Die Prozedur besteht aus 14 IF-Abfragen. Ohne Source-Clipping reduzieren sie sich auf acht. Das ist verschwindend gering gegenüber dem eigentlichen Blit-Vorgang, der Tausende von Bytes verschiebt. Er kann also guten Mutes jedesmal angewandt werden. Dennoch ist der vorgestellte Befehl um ein Vielfaches schneller als ClipBlit(). In Bild 1 sind die Abfragen grafisch dokumentiert.

Wer die Übergabe so vieler Parameter bei jedem Zeichenbefehl vermeiden möchte, kann dies durch globale Variablen erreichen. Da sich die Clip-Bereiche dw, dh, sw und sh meist nicht verändern, ist das sinnvoll. Ist jedoch sicher, daß der original Bildausschnitt niemals außerhalb des Bereichs liegt, kann vom Source-Clipping abgesehen werden.

Im Listing finden Sie keine Makros wie z.B. MIN(a,b) oder »? :« (bedingte Bewertung). Mit Absicht: Der Compiler generiert so besseren Code. Das soll kein Plädoyer für überkommenen Programmierstil sein, sondern notwendiger Kompromiß beim Schaffen zeitkritischen Codes.

Üblicherweise belegen wir bei BltBitMap den Parameter »minTerm« mit 0xC0 und »mask« mit 0xFF: so blitten wir alle Bitplanes. »tempA« darf null sein, wenn sich der Quell- und Zielbereich nicht überlappen. Ansonsten müssen wir einen Zeiger auf einen Speicherbereich für eine Bildschirmzeile (maximum 128 Byte) angeben. Tun wir das nicht, allokiert BltBitMap automatisch einen 128 Byte großen Speicherbereich, der allerdings nicht freigegeben wird.

Clippen von Linien

Clippen von Linien ist aufwendiger als das von Rechtecken. Es genügt nicht einfach, eine X-Koordinate auf Null zu setzen (sofern sie kleiner ist), während die zugehörige Y-Koordinate unverändert bleibt. Die Linie liegt zwar dann nicht mehr außerhalb des sichtbaren Bereichs, aber ihre Steigung ist falsch (Bild 2). Bei Modifikation einer Koordinate ist die zweite ebenfalls zu ändern.

Eine hocheffiziente Lösung dieses Problems lieferten schon in den sechziger Jahren Dan Cohen und Ivan Sutherland. Das im Anschluß vorgestellte Verfahren bezeichnet man daher auch als den Cohen-Sutherland-Algorithmus. Dahinter verbirgt sich folgende Idee: Man verlängert die Linien gedanklich bis über den Bildschirmrand hinaus (gestrichelte Linien in Bild 3). Insgesamt sind es acht Fortsetzungen, die durch Kombinationen von Oben, Unten, Rechts und Links entstehen. Im Bild finden Sie alle möglichen. Es gibt z.B. Linien, die völlig außerhalb des Bildschirms liegen. Von anderen wiederum ist mindestens ein Punkt sichtbar, oder sie streifen nur den sichtbaren Bereich. Selbstverständlich gibt's auch solche, die vollständig innerhalb liegen. Ein einziges Durcheinander.

Auf den ersten Blick scheinen die IF-Abfragen ins Unermeßliche zu steigen. Wie sieht also die Systematik aus, die, auf das Notwendigste reduziert, festlegt, wann eine Linie ganz oder teilweise gezeichnet oder vollständig verworfen wird?

Zuerst unterziehen wir beide Endpunkte einer Prüfung, in welchem Außenbereich seine Koordinaten liegen: Oben, Unten, Rechts oder Links (O, U, R bzw. L). Für jeden Punkt können maximal zwei Bereiche zutreffen. Der erste Punkt P1 der Linie A in Bild 3 liegt in R, P2 in U. Aber auch P1 von Linie B liegt in R, und P2 in U. Linie B jedoch liegt vollständig außerhalb des Bildschirms.

Folgendes läßt sich schon erkennen: Wenn sich weder P1 noch P2 in O,U,R, oder L verweilt, muß nicht geclippt werden – die Linie befindet sich vollständig innerhalb. Linie C demonstriert das.

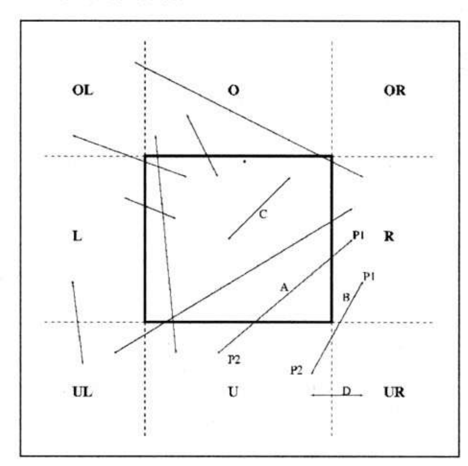


Bild 3: So ist es richtig – durch gedankliche Verlängerung der Linien und Aufteilung in Bereiche ist das 2-D-Clippen für uns kein Problem

Außerdem gilt: Haben sowohl P1 als auch P2 mindestens eine gleiche Seite (beide z.B. U (Linie D)), ist es nicht notwendig, diese zu zeichnen. Die Funktion CheckXY() prüft die Kantenlagen der Linien und setzt für jede der vier Bereiche O, U, R, bzw. L ein Bit in der Variablen K1 bzw. K2. Eine logische Und-Verknüpfung prüft alle Linien auf die Lage außerhalb des Bildes.

Treffen beide Bedingungen nicht zu, muß Kante für Kante überprüft und verkürzt wer-

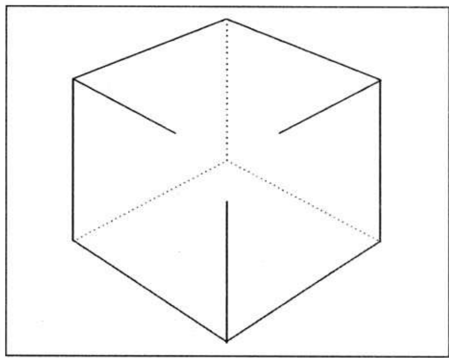


Bild 4: Mit großem CE-Wert ist der Würfel nahezu vollständig sichtbar – lediglich die vorderste Ecke taucht in die imaginäre Clip-Ebene ein

den. Nehmen wir an, ein Punkt liegt im R-Bereich (X >= XMax). Wir wissen, daß dann X auf XMax-1 zu setzen ist. Die Y-Strecke ist dann proportional der noch sichtbaren X-Strecke zu kürzen. Das geschieht mit:

```
y=y+(XMin-x)*dy/dx
```

wobei dy die Differenz der beiden Y-Koordinaten und dx die der beiden X-Koordinaten repräsentiert. Eine Division durch null ist unmöglich, da in diesem Fall (dx oder dy gleich null) beide Punkte auf derselben Seite liegen und die erste Abfrage

```
if((K1 & K2)!=0)
```

diese Eventualität abfängt.

y kann nach dieser Berechnung kleiner als YMin oder größer bzw. gleich YMax sein, wenn z.B. die Linie von R nach U gezeichnet wird und die untere rechte Ecke des Bildbereichs nicht schneidet. Es genügt also die Abfrage:

```
if(y<YMin || y>YMax)
  /* Linie kreuzt außerhalb die Felder */
  return(FALSE);
```

Das ist das ganze Geheimnis. Alle anderen Fallunterscheidungen sind Spiegelungen und Symmetrien dieser zwei Zeilen. Insgesamt sind vier Fallunterscheidungen pro X- bzw. Y-Koordinate nötig. Man muß sich vor Augen halten, daß von den vielen Programmzeilen in den meisten Fällen nur ganz wenige durchlaufen werden – bei Linien, die über den Rand des Bildschirms hinaus gezeichnet werden. Linien, bei denen das Clippen unnötig ist, handelt das Programm mit drei IF-Abfragen ab. Linien, die sowohl in X- als auch Y-Richtung zu beschneiden sind, benötigen im ungünstigsten Fall sieben Fallunterscheidungen und vier X/Y-Berechnungen (zzgl. der obligatorischen zwei bis vier

der Funktion CheckXY() pro Punkt). Schneller geht's also kaum – wenigstens nicht auf algorithmischem Weg. Der Quelltext des 2-D-Clippings besteht aus den Funktionen CheckXY(), MoveClip() und DrawClip(). MoveClip und DrawClip lassen sich wie die Funktionen Move() bzw. Draw() der graphics.library verwenden. Auf den RastPort wird via globaler Variable zugegriffen. Die Variablen x1, y1, dx und dy sind vom Typ LONG, da sonst Überläufe vorkommen.

3-D-Clipping

Warum 3-D-Clipping, wenn 2-D-Clipping bereits dafür sorgt, daß sich niemals Linien außerhalb des Bildschirms oder Fensters zeichnen lassen? Denken wir uns wieder einen dreidimensionalen Würfel als Drahtgittermodell. Jeder Eckpunkt des Würfels ist als Koordinate X,Y,Z gegeben. Die Formel für die Projektion auf den Bildschirm:

```
X'=PRF*X/(Z+PRF);
Y'=PRF*Y/(Z+PRF);
```

Z ist der Abstand vom Betrachter und PRF ein Projektionsfaktor, der den Grad der Zentralperspektive regelt. Je größer PRF, umso flacher wirkt die Perspektive. Man erkennt sofort, daß ein Objekt mit dem Abstand 0+PRF eine Division durch null zur Folge hätte, denn ein Objekt, das genau im Brennpunkt der Kamera (oder des Betrachters) liegt, wäre unendlich groß zu projizieren. Ein negativer Z-Wert bedeutet, daß sich das Objekt hinter der Kamera aufhält. In diesem Fall liefert die Projektion trotzdem positive X'/Y'-Werte, obwohl ein Objekt hinter der Kamera mit Sicherheit nicht zu sehen ist. 3-D-Clipping ist also nichts anderes als das Abschneiden aller Linien an einer Ebene, die senkrecht zur Blickrichtung irgendwo vor (im positiven Bereich) der Kamera liegt. Dieser Abstand ist wählbar und im folgenden mit CE (Clip-Ebene) bezeichnet.

Bild 4 zeigt einen schräg von vorn betrachteten Würfel bei groß gewähltem CE. Man erkennt, daß 3-D-Clipping nur für nahe gelegene, sehr groß projizierte Objekte notwendig ist.

Betrachten wir eine Linie, deren Anfang vor, das Ende hinter der Clip-Ebene liegt. Die Koordinaten des Punkts vor der Clip-Ebene sind zu transformieren. Z erhält den CE zugeordneten Wert, X bzw. Y müssen sich demzufolge proportional zum Streckenverhältnis der noch sichtbaren Linie verkürzen – jetzt allerdings im Raum und nicht in der Bildebene, wie beim 2-D-Clipping demonstriert. Die Vorgehensweise ist dennoch äquivalent. Bild 5 dokumentiert die die Clip-Ebene schneidende Linie und den daraus entstehenden Punkt mit den Koordinaten XC, YC und ZC. Er liegt genau auf der Clip-Ebene (ZC gleich CE).

Das Prinzip des 3-D-Clippings sollte jetzt klar sein. Es exisitiert jedoch nicht nur das hier gezeigte Verfahren, das zuerst im Raum und anschließend jede Koordinate nochmals in der Ebene isoliert: Es stehen Verfahren zur Verfügung, die geschickt im Raum clippen, das einen anschließenden 2-D-Algorithmus überflüssig macht.

Zumindest theoretisch ist sichergestellt, daß die projizierten Koordinaten niemals außerhalb des Bildschirms liegen. Das Verfahren basiert nun darauf, nicht nur alle

```
  1: /*
  2:  * Autor: Bernhard Emese
  3:  * Demo für 2d 3d Clipping Algorithmus: Rechteckiges Source-
  4:  * und Destinationclipping nach Cohen-Sutherland für Aztec-C
  5:  * Version 5.2a. Compileraufruf:
  6:  * cc cliptest.c -ps -f8 -c2
  7:  * Wer über keinen Floatingpoint-Prozessor und keinen
  8:  * MC68020/30-Prozessor verfügt, muß die Include-Datei
  9:  * <libraries/mathffp.h> einbinden und übersetzen mit
 10:  * cc cliptest.c -ps -ff
 11:  */
 12: #include <functions.h>
 13: #include <stdlib.h>
 14: #include <stdio.h>
 15: #include <exec/types.h>
 16: #include <intuition/intuition.h>
 17: #include <intuition/screens.h>
 18:
 19: #define WIDTH  640
 20: #define HEIGHT 512         // 512 für Interlace, sonst 256
 21: #define XMin 20            // erste erlaubte Koordinate links
 22: #define XMax (WIDTH-20)    // letzte erlaubte Koordinate rechts
 23: #define YMin 20            // erste erlaubte Koordinate oben
 24: #define YMax (HEIGHT-20)   // letzte erlaubte Koordinate unten
 25: #define OBEN   0x01        // Flags für 2-D-Clipping
 26: #define UNTEN  0x02
 27: #define LINKS  0x04
 28: #define RECHTS 0x08
 29: #define FALSE 0
 30: #define TRUE  1
 31: #define NULL 0L
 32:
 33: struct RastPort *RPort;
 34: LONG   x1,y1,x2,y2;      // Anfangs- und Endkoordinate der zu
 35:                          // zeichnenden 2d Linie
 36: SHORT  k1,k2;    // Kanten-Flags für Anfangs und Endpunkt
 37:
 38: void ClipBltBitMap(struct BitMap *SrcMap,SHORT sx,
 39:                    SHORT sy,struct BitMap *DstMap,
 40:                    SHORT dx,SHORT dy,SHORT sizex,SHORT sizey,
 41:                    SHORT minTerm,SHORT mask, SHORT *tempA,
 42:                    SHORT sw,SHORT sh,SHORT dw,SHORT dh)
 43: {
 44:   /* Source-Clipping -- das auszuschneidende Rechteck kann in
 45:    * allen Richtungen zu groß sein */
 46:   if(sx>=sw || sy>=sh)
 47:      return;   // gewünschter Ausschnitt existiert nicht
 48:
 49:   if(sx<0) {
 50:     /* wenn sx kleiner 0, dann sx=0 setzen und sizex
 51:      * verkleinern */
 52:     sizex+=sx; // sizex wird kleiner, da sx negativ
 53:     dx-=sx;    // dx entsprechend nach rechts verschieben
 54:     sx=0;
 55:   }
 56:   if(sy<0) {
 57:     /* Y wird genau wie X behandelt: sy=0 setzen und sizey
 58:      * verkleinern */
 59:     sizey+=sy; // sizey wird kleiner, da sy negativ
 60:     dy-=sy;    // dy entsprechend nach unten verschieben
 61:     sy=0;
 62:   }
 63:
 64:   /* Jetzt kann der Ausschnitt nur noch rechts und/oder unten
 65:    * über den Quellbereich hinausragen */
 66:
 67:   if(sizex > sw-sx)
 68:     sizex=sw-sx;
 69:
 70:   if(sizey > sh-sy)
 71:     sizey=sh-sy;
 72:
 73:   /* Destination-Clipping: Das zu zeichnende Recteck kann in
 74:    * allen Richtungen zu groß sein. Die Vorgehensweise ist
 75:    * zu dem des Source-Clippings identisch */
 76:
 77:   if(dx>=dw || dy>=dh)
 78:     return;   // Zeichenbereich liegt komplett außerhalb
 79:
 80:   if(dx<0) {
 81:      /* Wenn das Ziel zu weit links leigt, dann dx=0 setzen und
 82:       * sizex verkleinern */
 83:     sizex+=dx;
 84:     sx-=dx;     // sx wird größer
 85:     dx=0;       // am linken Rand wird geclippt
 86:   }
 87:   if(dy<0) {
 88:     sizey+=dy;  // sizey wird kleiner
 89:     sy-=dy;     // sy wird größer
 90:     dy=0;       // am oberen Rand wird geclippt
 91:   }
 92:
 93:   /* Ragt der Ausschnitt noch über den linken und/oder unteren
 94:    * Bildschirmrand ? */
 95:
 96:   if(sizex > dw-dx)
 97:     sizex=dw-dx;      // sizex verkleinern
 98:
 99:   if(sizey > dh-dy)
100:     sizey=dh-dy;      // sizey verkleinern
101:
102:   /* Jetzt kann durch Abschneiden an allen Ecken und Enden von
103:    * sizex und sizey evtl. nichts mehr übrig sein */
104:
105:   if(sizex>0 && sizey>0)
106:     BltBitMap(SrcMap,sx,sy,DstMap,dx,dy,
107:               sizex,sizey,0xc0L,0xffL,NULL);
108: }
109:
110: SHORT CheckXY (LONG x,LONG y) {
111:   SHORT k;
112:
113:   if(y<YMin)          // Liegt Y Koordinate unter Minimum ?
114:     k=UNTEN;
115:   else if(y>YMax)     // Wenn nicht, kann sie nur Oben liegen
116:     k=OBEN;
117:   else
118:     k=0;              // Wenn nicht Oben, dann im Bild
119:
120:   if(x<XMin)             // Liegt X-Koordinate unter Minimum ?
121:     return(k|=LINKS);    // Return, da X nicht gleichzeitig
122:                          // Links und Rechts liegen kann
123:   else if(x>XMax)        // Wenn nicht, dann Rechts
124:     return(k|=RECHTS);
125:   return(k);             // Oder in Bildmitte
126: }
127:
128: void MoveClip (LONG x,LONG y) {
129:   // setze Flags je nach Lage des Punkts in O, U, R, oder L
130:   k1=CheckXY(x1=x,y1=y);
131: }
132:
133: SHORT DrawClip(LONG x2c,LONG y2c) {
134:   LONG  x1c,y1c,dx,dy;
135:   SHORT k;
136:
137:   k2=CheckXY(x2=x2c,y2=y2c);   // setze Flags
138:
```

```
139:   x1c=x1; y1c=y1; k=k1;
140:
141:   /* x1c,y1c mit letzter nicht-geclippter Koordinate
142:    * x1,y1 laden */
143:
144:   x1=x2c; y1=y2c; k1=k2;
145:
146:   // ungeclippte neue Koordinate für nächstes Draw abspeichern
147:
148:   if((k & k2)!=0)
149:     return(FALSE);  // Beide Punkte liegen auf gleicher Seite
150:                     // außerhalb
151:   dx=x2c-x1c;
152:   dy=y2c-y1c;
153:   if(k) {             // Liegt P1 irgendwo außerhalb ?
154:     if(k & LINKS) {  // wenn links, dann
155:       y1c+=(XMin-x1c)*dy/dx;   // dx kann hier nie 0 sein
156:       if(y1c<YMin || y1c>YMax)
157:         return(FALSE);  // Linie kreuzt außerhalb die Felder
158:       x1c=XMin;
159:     } else if(k & RECHTS) {
160:       y1c+=(XMax-x1c)*dy/dx;
161:       if(y1c<YMin || y1c>YMax)
162:         return(FALSE);   // Linie kreuzt außerhalb die Felder
163:       x1c=XMax;
164:     }
165:     if(k & UNTEN) {
166:       x1c+=(YMin-y1c)*dx/dy;  // dy kann hier nie 0 sein
167:       if(x1c<XMin || x1c>XMax)
168:         return(FALSE);   // Linie kreuzt außerhalb die Felder
169:       y1c=YMin;
170:     } else if(k & OBEN) {
171:       x1c+=(YMax-y1c)*dx/dy;
172:       if(x1c<XMin || x1c>XMax)
173:         return(FALSE);   // Linie kreuzt außerhalb die Felder
174:       y1c=YMax;
175:     }
176:   }
177:   if(k2) {                // liegt P2 irgendwo außerhalb
178:     if(k2 & LINKS) {
179:       y2c+=(XMin-x2c)*dy/dx;   // dx kann hier nie 0 sein
180:       if(y2c<YMin || y2c>YMax)
181:         return(FALSE);   // Linie kreuzt außerhalb die Felder
182:       x2c=XMin;
183:     } else if(k2 & RECHTS) {
184:       y2c+=(XMax-x2c)*dy/dx;
185:       if(y2c<YMin || y2c>YMax)
186:         return(FALSE);   // Linie kreuzt außerhalb die Felder
187:       x2c=XMax;
188:     }
189:     if(k2 & UNTEN) {
190:       x2c+=(YMin-y2c)*dx/dy;   // dy kann hier nie 0 sein */
191:       if(x2c<XMin || x2c>XMax)
192:         return(FALSE);   // Linie kreuzt außerhalb die Felder
193:       y2c=YMin;
194:     } else if(k2 & OBEN) {
195:       x2c+=(YMax-y2c)*dx/dy;
196:       if(x2c<XMin || x2c>XMax)
197:         return(FALSE);   // Linie kreuzt außerhalb die Felder
198:       y2c=YMax;
199:     }
200:   }
201:   Move(RPort,x1c,y1c);
202:   Draw(RPort,x2c,y2c);
203:   return(TRUE);
204: }
```

Linien gegen die senkrecht zur Blickrichtung liegende Clip-Ebene zu überprüfen, sondern zusätzlich an den vier Flächen eines nach hinten in die Unendlichkeit reichenden Pyramidenstumpfs. Stellt man sich einen Blick in den Raum vor, der bei einer 3-D-Zeichnung auf dem Bildschirm einsehbar ist, ist das genau der Inhalt jenes Pyramidenstumpfs und folglich klar, daß ein Raum-Clipping, welches Objekte nur in diesem zuläßt, auf das nachträgliche 2-D-Clipping verzichten kann.

Das aber ist Theorie. In der Praxis kämpft man bei dieser Projektion mit Ungenauigkeiten der Fließkomma-Arithmetik: ein Punkt Ungenauigkeit führt dazu, daß über den sichtbaren Bereich hinausgezeichnet wird und bedeutet oftmals den Systemabsturz. 2-D-Clipping hingegen ist ein auf Ganzzahlen operierendes Verfahren und schließt ebendiese Fehler aus. Darüber hinaus ist die Berechnung des vollständigen 3-D-Clippings wesentlich zeitintensiver und rechenaufwendiger als das hier vorgestellte, da einer Ebenenberechnung mit 2-D-Clipping fünf 3-D-Berechnungen (vordere Clip-Ebene und vier Pyramidenseiten) gegenüberstehen.

Unser Programm prüft also jede Linie gegen die Clip-Ebene und schneidet sie gegebenenfalls ab. Ab einer definierten Stelle enden die Linien irgendwo im Raum, als rückte man mit einer Zange dem Drahtgittermodell zu Leibe und zwickte einige Drähte ab. Wählt man also die Clip-Ebene nahe dem Betrachter, sind die Drahtenden niemals sichtbar.

Unser Listing demonstriert das auf anschauliche Weise. Der Würfel wird nach rechts, links, oben und unten bewegt, dann vor und zurück, wobei Teile des Drahtgitters aus dem Bild verschwinden oder in die Clip-Ebene eintauchen. Das Resultat des Clippings ist sichtbar und überprüfbar.

Die hier vorgestellte Mini-3-D-Library ist bereits dazu ausgelegt, Objekte im Raum zu drehen, denn auch bei einer Drehung des Würfels ohne Verschiebung des Betrachterstandpunkts ist es wahrscheinlich, daß der Würfel bereits die Clip-Ebene schneidet, selbst wenn er es in seiner Ruhelage noch nicht tut. Deshalb gehen wir kurz auf die vom Programm verwendete Rotationsmatrix ein.

Sie ist ein Schlüssel für viele 3-D-Transformationen, nicht nur für Drehungen. Die vier Parameter für jede Koordinate besitzen eine Fülle von Kombinationen, die hier zu erörtern unseren Rahmen sprengen würde. Es lassen sich jedoch mit einer solchen Matrix Verschiebungen, Verzerrungen, Skalierungen, asymmetrische Drehungen etc. vornehmen, vorausgesetzt, man hat die Werte vorher sinnvoll besetzt. Die im Programm vorgegebene ZeroMatrix ist übrigens das neutrale Element der Matrizenmultiplikation, führt also keine Transformation aus.

In unserem Beispiel besetzt die Funktion Rotate() die Matrix nur mit Werten für die drei Drehungen im Raum. Man gibt drei Winkel vor, um die jeder Eckpunkt des Würfels rotiert werden soll. Hat man einmal die Matrix mit gültigen Werten besetzt, wird sie für jeden Eckpunkt berechnet. Die Vorberechnung ist nicht ganz einfach und insofern wichtig, da die Transformationsfunktionen ansonsten für alle acht Eckpunkte mit Hilfe der trigonometrischen Funktionen Sinus und Cosinus berechnet werden müßten. Für einen Würfel entspricht die Ersparnis also 8:1, bei komplexeren Objekten sogar noch günstiger.

Ideal ist die Matrix dann, wenn sich nicht das Objekt, sondern der Standpunkt des Betrachters ändert (z.B. ein Kameraschwenk), da in diesem Fall die Koordinaten aller Objekte zu berechnen sind. Unser Listing enthält allerdings keine Kameratransformation, es »schaut« stur in die Z-Richtung.

Die einzige Kameratransformation ist die freie Wahl des Betrachterstandpunkts mit BX, BY und BZ. Die Wirkung eines größeren Abstands durch die verkleinerte Darstellung des Objekts erzielt der Betrachter dadurch, indem sein Wert von jedem Punkt subtrahiert wird

Mit dem beschriebenen Verfahren lassen sich alle linienorientierten Objekte ordnungsgemäß und in der Praxis ausreichend effizient clippen. Dennoch sind die Anforderungen moderner Computergrafik keineswegs erschöpt. Zu den am häufigsten verlangten Verfahren zählt das Polygon-Clipping. Polygone (geschlossene Linienzüge), die über den 2-D-Bereich hinausragen, können zwar Linie für Linie gekappt werden – das Ergebnis ist aber kein geschlossenes Polygon mehr.

Möchte man ein so verstümmeltes Polygon mit einer Farbe füllen – Sie können sich vorstellen, was passiert: Da die Linien zum Bildschirmrand nicht mehr geschlossen sind, »läuft« die Farbe sozusagen aus. Doch auch für dieses Problem gibt's Lösungen. Diese gehen nach dem Prinzip vor, ein aus wenig Punkten bestehendes Polygon mit mehr Punkten darzustellen. Der Vorteil: die Kantenlänge der Linien wird verkürzt.

Außerdem ist eine zusätzliche Fläche einzuführen, falls es sich um einen massiven Würfel (kein Hohlwürfel) handelt, die den Blick in den Würfelinnenraum verdeckt. Flächen- und kantenorientiertes 3-D-Zeichnen von Objekten ist nicht immer trivial.

Das hier abgedruckte Listing finden Sie auch auf der Diskette zum Heft (Seite 114). Das erspart Tipparbeit und unnötige Fehlersuche. *rz*

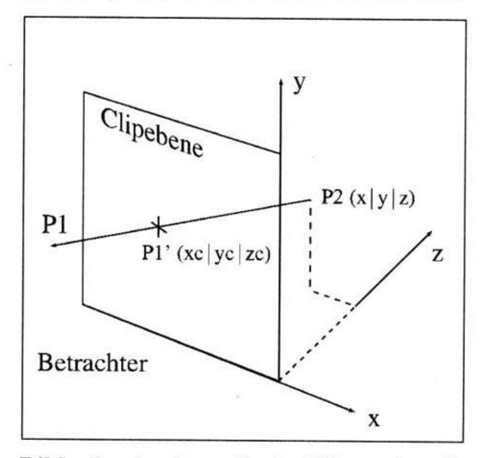


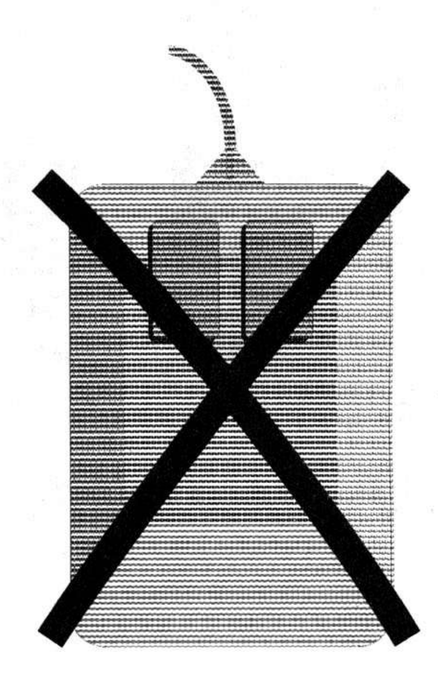
Bild 5: Auch auf dreidimensionale Objekten läßt sich der zweidimensionale Clipping-Algorithmus anwenden

```
205:
206: // Anfangs- und Endpunkte der 3-D-Linie
207: LONG X1,Y1,Z1,X2,Y2,Z2;
208: LONG    PRF;
209: SHORT  CE;          // Abstand der ClipEbene vom Brennpunkt
210: SHORT  BX,BY,BZ; // Betrachterstandpunkt (Kamera)
211:
212: FLOAT ZeroMatrix[4][4] = {
213:   {1,0,0,0}, {0,1,0,0}, {0,0,1,0}, {0,0,0,1} };
214: FLOAT Matrix[4][4];
215:
216: #define CONVERTX(x,z)     (PRF*(x)/((z)+PRF)+WIDTH/2)
217: #define CONVERTY(y,z)     (PRF*(y)/((z)+PRF)+HEIGHT/2)
218:
219: void Transform3d (FLOAT M[4][4],LONG *X,LONG *Y,LONG *Z) {
220:   LONG XU,YU,ZU;
221:
222:   XU=*X;       // zu drehende verschiebende Koordinate
223:                // zwischenspeichern
224:   YU=*Y;
225:   ZU=*Z;
226:
227:   /* beliebige Tranformation ausführen */
228:   *X=M[0][0]*XU+M[1][0]*YU+M[2][0]*ZU+M[3][0];
229:   *Y=M[0][1]*XU+M[1][1]*YU+M[2][1]*ZU+M[3][1];
230:
231:   /* die Art der Drehung, Verschiebung etc. ist vom
232:    * Inhalt der Matrix abhängig */
233:   *Z=M[0][2]*XU+M[1][2]*YU+M[2][2]*ZU+M[3][2];
234: }
235:
236: void Move3dClip (LONG X,LONG Y,LONG Z) {
237:   Transform3d(Matrix,&X,&Y,&Z);
238:
239:   X1=X-BX;    // Betrachterstandpunkt von Koordinate abziehen
240:   Y1=Y-BY;
241:   Z1=Z-BZ;
242: }
243:
244: void Draw3dClip (LONG X,LONG Y,LONG Z) {
245:   FLOAT RQ;
246:
247:   Transform3d(Matrix,&X,&Y,&Z);
248:
249:   X=X-BX;      // Betrachterstandpunkt von Koordinate abziehen
250:   Y=Y-BY;
251:   Z=Z-BZ;
252:
253:   if(Z1>=CE) {          // alte Koordinate hinter Clipebene ?
254:     MoveClip(CONVERTX(X1,Z1),CONVERTY(Y1,Z1));
255:     if(Z>=CE)
256:       // neue Koordinate auch: kein Clipping erforderlich
257:       DrawClip(CONVERTX(X,Z),CONVERTY(Y,Z));
258:     else {
259:       // Zielkoordinate clippen, Streckenverhältnis berechnen
260:       RQ=(FLOAT)(CE-Z1)/(Z-Z1);
261:       X1+=(X-X1)*RQ;          // X proportional verkürzen
262:       Y1+=(Y-Y1)*RQ;          // Y proportional verkürzen
263:       Z1=CE;                  // Z auf Clipebene setzen
264:       DrawClip(CONVERTX(X1,Z1),CONVERTY(Y1,Z1)); // Zeichnen
265:     }
266:   } else              // alte Koordinate vor Clipebene
267:   if(Z>=CE) {   // neue Koordinate hinter Clipebene --> clippen
268:     RQ=(FLOAT)(CE-Z1)/(Z-Z1);
269:     X1+=(X-X1)*RQ;
270:     Y1+=(Y-Y1)*RQ;
271:     Z1=CE;
272:     MoveClip(CONVERTX(X1,Z1),CONVERTY(Y1,Z1));
273:     DrawClip(CONVERTX(X,Z)  ,CONVERTY(Y,Z));
274:   }
275:   X1=X; // neuen Punkt ungeclippt für nächstes Draw3d speichern
276:   Y1=Y;
277:   Z1=Z;
278: }
279:
280: #ifndef PI
281: #define PI  ((FLOAT) 3.141592653589793)
282: #endif
283:
284: void Rotate(FLOAT M[4][4],FLOAT A,FLOAT B,FLOAT C)
285: {
286:   FLOAT MSIN,SIN,COS,o;
287:
288:   movmem((char *)ZeroMatrix,
289:          (char *)Matrix,(int)4*4*sizeof(FLOAT));
290:
291:   SIN = sin(A=A*PI/180); MSIN = -SIN;
292:   COS = cos(A);
293:   o = M[0][1]*COS + M[0][2]*SIN;
294:   M[0][2] = M[0][1]*MSIN + M[0][2]*COS;
295:   M[0][1] = o;
296:   o = M[1][1]*COS + M[1][2]*SIN;
297:   M[1][2] = M[1][1]*MSIN + M[1][2]*COS;
298:   M[1][1] = o;
299:   o = M[2][1]*COS + M[2][2]*SIN;
300:   M[2][2] = M[2][1]*MSIN + M[2][2]*COS;
301:   M[2][1] = o;
302:   o = M[3][1]*COS + M[3][2]*SIN;
303:   M[3][2] = M[3][1]*MSIN + M[3][2]*COS;
304:   M[3][1] = o;
305:
306:   SIN = sin(B=B*PI/180); MSIN = -SIN;
307:   COS = cos(B);
308:   o = M[0][0]*COS + M[0][2]*MSIN;
309:   M[0][2] = M[0][0]*SIN + M[0][2]*COS;
310:   M[0][0] = o;
311:   o = M[1][0]*COS + M[1][2]*MSIN;
312:   M[1][2] = M[1][0]*SIN + M[1][2]*COS;
313:   M[1][0] = o;
314:   o = M[2][0]*COS + M[2][2]*MSIN;
315:   M[2][2] = M[2][0]*SIN + M[2][2]*COS;
316:   M[2][0] = o;
317:   o = M[3][0]*COS + M[3][2]*MSIN;
318:   M[3][2] = M[3][0]*SIN + M[3][2]*COS;
319:   M[3][0] = o;
320:
321:   SIN = sin(C=C*PI/180); MSIN = -SIN;
322:   COS = cos(C);
323:   o = M[0][0]*COS + M[0][1]*SIN;
324:   M[0][1] = M[0][0]*MSIN + M[0][1]*COS;
325:   M[0][0] = o;
326:   o = M[1][0]*COS + M[1][1]*SIN;
327:   M[1][1] = M[1][0]*MSIN + M[1][1]*COS;
328:   M[1][0] = o;
329:   o = M[2][0]*COS + M[2][1]*SIN;
330:   M[2][1] = M[2][0]*MSIN + M[2][1]*COS;
331:   M[2][0] = o;
332:   o = M[3][0]*COS + M[3][1]*SIN;
333:   M[3][1] = M[3][0]*MSIN + M[3][1]*COS;
334:   M[3][0] = o;
335: }
336:
337: struct Library *IntuitionBase,*GfxBase;
338: struct Window  *Window;
339: static struct NewWindow nw = {
340:   0,0, WIDTH,HEIGHT, 0,1, 0, ACTIVATE, NULL, NULL,
341:   (UBYTE *)"2d  3d Clipping", NULL, NULL, 0,0,WIDTH,HEIGHT,
342:   WBENCHSCREEN,
343: };
344:
345: /*
346:  * Zeichnet alle Kanten eines Drahtgitterwürfels und
347:  * löscht vorher den Hintergrund
348:  */
349: void Cube() {
350:   SetAPen(RPort,0);
351:   RectFill(RPort,XMin,YMin,XMax,YMax);
352:   SetAPen(RPort,1);
353:   Move3dClip(-250,+250,-250); Draw3dClip(+250,+250,-250);
354:   Draw3dClip(+250,-250,-250); Draw3dClip(-250,-250,-250);
355:   Draw3dClip(-250,+250,-250); Draw3dClip(-250,+250,+250);
356:   Draw3dClip(+250,+250,+250); Draw3dClip(+250,-250,+250);
357:   Draw3dClip(-250,-250,+250); Draw3dClip(-250,+250,+250);
358:   Move3dClip(+250,+250,-250); Draw3dClip(+250,+250,+250);
359:   Move3dClip(+250,-250,-250); Draw3dClip(+250,-250,+250);
360:   Move3dClip(-250,-250,-250); Draw3dClip(-250,-250,+250);
361:   Delay(2);
362: }
363:
364: Movement(SHORT i) {
365:   if(i<100) return(-i);       // fährt zuerst nach links
366:   if(i<300) return(-200+i); // dann nach rechts
367:   return(400-i);              // dann wieder auf den Anfangspunkt
368: }
369:
370: main() {
371:   SHORT i,j;
372:
373:   IntuitionBase=OpenLibrary((UBYTE *)"intuition.library",NULL);
374:   GfxBase=OpenLibrary((UBYTE *)"graphics.library",NULL);
375:
376:   Window=OpenWindow(&nw);  // öffnet Workbench-Fenster
377:
378:   RPort=Window->RPort;
379:
380:   SetAPen(RPort,2);
381:
382:   // Zeichnet eine Umrahmung, die nicht überschritten wird.
383:   Move(RPort,XMin-1,YMin-1);
384:   Draw(RPort,XMax+1,YMin-1);
385:   Draw(RPort,XMax+1,YMax+1);
386:   Draw(RPort,XMin-1,YMax+1);
387:   Draw(RPort,XMin-1,YMin-1);
388:
389:   PRF=1000; // Werte für 3d Clipping
390:   CE= 740;  // Clipebene ist 10 Einheiten vor dem Objekt
391:   BZ=-1000; // Betrachter ist 1000 Einheiten entfernt
392:
393:   for(i=0;i<=400;i+=4) {
394:     // Würfeldrehungen in X,Y,Z
395:     Rotate(Matrix,Movement(i),Movement(i),Movement(i));
396:     Cube();
397:   }
398:   Delay(50);
399:
400:   for(i=0;i<=400;i+=4) {
401:     BX=Movement(i)*10;
402:     // Würfel rollt nach rechts und links
403:     Rotate(Matrix,0,0,Movement(i));
404:     Cube();
405:   }
406:   Delay(50);
407:
408:   for(i=0;i<=400;i+=4) {
409:     BY=Movement(i)*10;
410:     // Würfel rollt nach oben und unten
411:     Rotate(Matrix,-Movement(i),0,0);
412:     Cube();
413:   }
414:   Delay(50);
415:
416:   for(i=0;i<=400;i+=4) {
417:     BZ=Movement(i)*10-1000;
418:     // Würfel rollt nach vorn und hinten
419:     Rotate(Matrix,-Movement(i),0,0);
420:     Cube();
421:   }
422:   Delay(50);
423:
424:   CloseWindow(Window);
425:   CloseLibrary(GfxBase);
426:   CloseLibrary(IntuitionBase);
427: }
428:
```

»Clipping.c«: Das Listing demonstiert das 2- und-3-D-Clipping und läßt sich weiter ausbauen

Lichtgriffel am Amiga

Eierlegende Wollmilchsau



Der Light-Pen (deutsch: Lichtgriffel) findet mit Abstand am wenigsten Verwendung am Gameport des Amiga. Das hat seinen Grund: Das Betriebssystem ist nämlich ein Gegner des Light-Pens und man muß es erst überlisten. Warum das so ist und wie man es macht, zeigen wir hier.

von Bernhard Emese

Den Amiga-Entwicklern sagt man nach, sie hätten eine »Eierlegende Wollmilchsau« entworfen. Jedenfalls haben sie versucht, einen »Joyballpaddelnden AnalogPotentioTrackStick« oder eine »Lichtblitzgriffelnde Rollmichmaus« in die Gameports einzuhängen. Tatsächlich muß man ihnen Respekt zollen. Was man alles ungeniert dort vorne in die neunpoligen Gameport-Buchsen des Amiga 'reinstecken kann, ohne daß Rauchwolken einen Racheakt des Computers anzeigen, ist schon erstaunlich.

Die Pins sind auf raffinierte Weise gemultiplext, abgeblockt, gepull-upped und gewissermaßen ineinander verschachtelt, so daß sich an jeden Gameport vier grundsätzlich verschiedene Peripheriegeräte anschließen lassen: Maus, Joystick, Proportional-Controller und Light-Pen (letzterer ist immerhin nur an einem Gameport anschließbar). Die Software kann beide Gameports auf unterschiedliche Weise abfragen: Einmal als Mouse-, als Joystick-Port und als Proportional-Port, oder eben als Light-Pen-Port. Entgegen der Dokumentation im Hardware-Reference-Manual [1] ist der Lichtgriffel nur an den zweiten Port des Amiga 2000 anschließbar. Die Pin-Belegung des 9poligen D-SUB-Steckers als Light-Pen-Port finden Sie in Bild 1.

Man erkennt, daß nur wenige Signale benötigt werden. Der Schalter, der in der Lichtgriffelspitze untergebracht ist, wird laut Dokumentation auf Pin 5 abgefragt. Man kann ohne weiteres einen Lichtgriffel des C-64 an den Amiga anschließen, den man überall preisgünstig erstehen kann. Der Schalter dieses Light-Pens allerdings ist über Pin 3 (Joystick Signal Links) abzulesen.

Erstaunlich, wie leicht das Skizzieren mit einem Lichtgriffel fällt, z.B. beim Zeichnen eines Kreises. Mit der Maus hat man ja schon Mühe, den Linienanfang genau zu treffen. Man müßte allerdings über einen fast waagerecht in den Schreibtisch eingelassenen flachen Monitor verfügen, um längeres Arbeiten mit dem Lichtgriffel zu ermöglichen. Zudem ist es mit der Auflösung und Pixelgenauigkeit nicht weit her. Horizontal beträgt die Auflösung nur 160 Pixel, unabhängig vom eingestellten Bildschirmmodus. Vertikal sind es immerhin so viele, wie Rasterzeilen existieren – im PAL-Interlace-Modus also 512. Mit Flickerfixer funktioniert er überhaupt nicht.

Das abgedruckte Programm ist der Überschaubarkeit wegen nur für einen Non-Interlaced-Bildschirm ausgelegt. Für Interlaced-Modi ist die zweite (Short Frame) Copper-Liste zu patchen und die Berechnung der Vertikalposition mit zwei zu multiplizieren. Jedenfalls gelingt es mit dem Lichtgriffel wesentlich besser, seinen Namen auf den Bildschirm zu pinseln, als das mit der teuersten Maus möglich wäre. Die ideale Anwendung für einen Lichtgriffel ist demzufolge nicht das Zeichnen, sondern die Auswahl von Menüfeldern und Gadgets.

Wie funktioniert ein Lichtgriffel? Auf den ersten Blick grenzt es schon fast an Zauberei, wenn man mit einem Stift über den Bildschirm fährt und dort zeichnen kann, als wäre es eine Schiefertafel. Woher sieht denn der Bildschirm, wo sich der Lichtgriffel befindet? Wieso kann der Bildschirm überhaupt etwas sehen?

Das Prinzip ist simpel: Der Custom-Chip Agnus, verantwortlich für das Video-Timing des Amiga, enthält zwei Register, die sowohl die augenblickliche vertikale als auch horizontale Position des Elektronenstrahls beinhalten. Sie werden normalerweise permanent inkrementiert und zu Beginn eines neuen Durchlaufs wieder auf Null gesetzt (Bild 2).

Die unteren 8 Bit von VHPOS enthalten die aktuelle horizontale Strahlposition in einer Auflösung von zwei Lowres- oder vier Hires-Pixeln (1/160 des Bildschirms), da H0 fehlt. Der Wert für die horizontale Position kann zwischen \$00 und \$e3 (0 bis 227 dezimal) liegen. Die horizontale Rücklaufperiode liegt zwischen \$0f bis \$35. Die oberen 8 Bit zusammen mit dem Bit 0 von VPOS geben die aktuelle vertikale Strahlposition an, diesesmal in der vollen Auflösung von einer Rasterzeile (pro Halbbild 0 bis 312, also 625 Zeilen in PAL). Die vertikale Rücklaufperiode reicht von 0 bis 25. Zusätzlich enthält VPOS noch 1 Bit, das im Interlaced-Modus bei jedem ungeraden Halbbild gesetzt ist.

	\$dff004	\$dff006		\$dff004	\$dff006
Bit	**VHPOS**	**VPOS**	**Bit**	**VHPOS**	**VPOS**
15	V7	LOF	7	H8	--
14	V6	--	6	H7	--
13	V5	--	5	H6	--
12	V4	--	4	H5	--
11	V3	--	3	H4	--
10	V2	--	2	H3	--
9	V1	--	1	H2	--
8	V0	--	0	H1	V8

Bild 2: Die Register VHPOS und VPOS des Custom-Chips Agnus – sie werden vom Amiga permanent inkrementiert (erhöht)

①②③④⑤ ⑥⑦⑧⑨

Pin	Light-Pen	Joystick
1	unbenutzt	Oben
2	unbenutzt	Unten
3	Schalter	Links
4	unbenutzt	Rechts
5	(Schalter)	unbenutzt
6	LP-Signal	Feuerknopf
7	+5 Volt	+5 Volt
8	GND	GND
9	unbenutzt	unbenutzt

Bild 1: Die Anschlußbelegung des Amiga-Gameports und die Bedeutung der einzelnen Pins sowohl für den Light-Pen als auch für den Joystick

Ein Lichtgriffel besteht aus einem Fototransistor mit einer kleinen Schmitt-Trigger-Schaltung, die in dem Moment, in dem der Elektronenstrahl am Fototransistor des Lichtgriffels vorbeisaust, einen Impuls erzeugt. Der Lichtgriffeleingang des Amiga reagiert auf diesen Impuls und stoppt augenblicklich das Beamcounter-Register. Dieses läßt sich danach via Software auslesen. Es beinhaltet die Position, in der sich der Lichtgriffel befinden muß. Die Pixel-Position wird durch

Abzug eines Offsets aus der Strahlposition errechnet. Erst bei Beginn des nächsten Bildes (nach Ende der vertikalen Austastlücke, also ab Zeile 26) wird der Zähler erneut gestartet. So läßt sich mit einem Lichtgriffel bis zu 50mal pro Sekunde eine Position ermitteln und Linien oder ähnliches auf den Bildschirm zaubern.

Die Stoppmöglichkeit des Beamcounters ist normalerweise abgeschaltet, sonst würde auch bei einfachem Mausbetrieb ohne Lichtgriffel die Betätigung der Maus oder eines Joysticks am Port 2 das Register VPOSR am Mitzählen hindern. Das sollte aber zur Feststellung der aktuellen Rasterstrahlposition jederzeit zur Verfügung stehen, z.B zum synchronisierten Blitten ohne Flimmereffekte. Für den Betrieb eines Lichtgriffels ist deshalb das LPEN-Bit (Bit 3 von BPLCON0) auf High zu setzen.

Da das Betriebssystem keinerlei Softwareunterstützung zum Betrieb eines Lichtgriffels liefert, ist alles zu Fuß zu programmieren – sogar die Copper-Liste muß gepatcht werden, da sie sonst das LPEN-Bit alle 50stel Sekunde wieder auf auf Null setzt.

Jeder Lichtgriffel hat außer dem lichtempfindlichen Sensor noch einen Schalter in der Schreibspitze, der geschlossen wird, wenn man den Bildschirm berührt und Druck ausübt. Dieser Schalter wird vom Gameport ebenfalls abgefragt, wobei in offenem Zustand normalerweise keine Aktion ausgeführt wird. Die Light-Pen-Software kann leicht zwischen drei Zuständen unterscheiden:

❍ Der Lichtgriffel hat ausgelöst (Elektronenstrahl wurde registriert)
❍ Der Lichtgriffel hat nicht ausgelöst (Lichtgriffel ist zuweit weg)
❍ Der Schalter ist gedrückt (meist hat dann der Lichtgriffel auch ausgelöst)

Es gibt nun noch ein paar Tücken: Der Elektronenstrahl ist nur dort zu registrieren, wo der Bildschirm nicht ganz dunkel ist. Es genügt, als Hintergrundfarbe Grau einzustellen. Man kann aber auch absichtlich Teile des Bildschirms für den Lichtgriffel ausmaskieren, indem man dort einfach schwarze Bereiche zeichnet. Vorsicht ist auch bei Verwendung der Farbe Rot geboten. Die meisten Fotodioden und -transistoren in Lichtgriffeln registrieren kein Rot, also bleiben auch rote Bereiche unerkannt. Meistens klickt man mit der lichtempfindlichen Spitze jedoch auf ein Objekt, das hell auf dunkel abgebildet ist.

Ob der Light-Pen ausgelöst wurde, ist relativ unkompliziert festzustellen. Man liest die Strahlposition zweimal hintereinander aus den Registern. Sind die Werte identisch und liegen sie innerhalb des sichtbaren Bereichs, steht der Zähler still, der Light-Pen hat also ausgelöst. Noch einfacher ist es, während des Vertical-Blanking-Interrupts ($0 < V < 25$). Man liest die Register einmal. Liegt der Wert innerhalb des sichtbaren Bildes, ist er gültig.

Bleibt die Feststellung, daß der Lichtgriffel zum Zeichnen zwar zu grob, zur Gadget-Auswahl direkt am Bildschirm und für Programme, die ohne Maus und ohne Tastatur auskommen, ideal geeignet ist. *rz*

Literaturhinweise

[1] Commodore Amiga, AMIGA ROM Kernel Reference Manual Hardware, Third Edition, Reading 1991


```
 1: /*
 2:  * LightPen Programm für Amiga
 3:  * Autor:Bernhard Emese
 4:  * Hier wird gezeigt, wie auf dem AMIGA eine Lightpen-Software
 5:  * aussehen kann. Alles, was man dafür braucht, ist der richtige
 6:  * Umgang mit den Beamposition Registern VPOS und VHPOS.
 7:  * Der C-Source ist mit Aztec 5.0 lauffähig und kann leicht
 8:  * an LATTICE oder andere angepaßt werden.
 9:  * Aufruf: cc lpen.c -ps
10:  *         ln lpen.o -lLIB:/c16
11:  * Sobald das Programm gestartet ist, kann man auf dem
12:  * Workbench-Screen in den Workbench-Farben mit einem Lichtgriffel
13:  * Linien zeichnen. Das Programm geht von einem Non-Interlaced
14:  * Workbench-Screen aus. Mit Preferences muß die Hintergrundfarbe
15:  * auf eine mindestens mittlere Helligkeit gesetzt werden, da
16:  * sonst der Lichtgriffel nicht anspricht. Vorsicht auch bei Farbe
17:  * Rot. Manche Griffel können dann nichts »sehen«.
18:  * Leider funktioniert der Lichtgriffel nicht mit Flickerfixer-
19:  * Karten. Die verzögerte Ausgabe der Bildinformation führt dort
20:  * natürlich zu völlig falschen Positionswerten. Außerdem wird
21:  * durch das Patchen der Copperliste die Anzeige des Bildschirms
22:  * beeinflußt. Beim Amiga 3000 muß der Schalter des integrierten
23:  * Flickerfixers in Stellung aus gebracht werden.
24:  */
25: #include <functions.h>
26: #include <exec/types.h>
27: #include <intuition/intuitionbase.h>
28: #include <intuition/intuition.h>
29: #include <hardware/custom.h>
30: #include <graphics/gfxbase.h>
31:
32: /***  Defines ***/
33: #define LEFT_BUTTON 0x40  /* Linker Mausknopf am CIA-Port */
34: #define DXOFFSET 20       /* Zum Justieren des X-Werts */
35: #define PENDOWN    (1<<9) /* für VC-64 Lichtgriffel */
36: #define BEAMDETECT (1<<0) /* interne Bits der LPen Routine */
37: #define PENPRESSED (1<<1)
38: #define WIDTH 640         /* Grösse des Zeichenwindows */
39: #define HEIGHT 256
40: #define IDCMPFLAGS  NULL  /* keine IDCMP-Messages */
41: #define WINDOWFLAGS (ACTIVATE | BORDERLESS)  /* Ohne Border */
42: /* Hardwareregister */
43: #define custom (*((struct Custom *)0xdff000))
44:
45: /*** Prototypes ***/
46: void   InitLPen(struct View *View);
47: USHORT LPen(struct View *View);
48:
49: /*** Variablen ***/
50: struct IntuitionBase *IntuitionBase;
51: struct GfxBase *GfxBase;
52: struct View      *LPenView;
53: struct RastPort *LPenPort;
```

```
 54: struct Window   *LPenWindow;
 55: SHORT PenX,PenY,OldPenX,OldPenY,OldPen;
 56: USHORT *Joy0Dat=(USHORT *)0xdff00a,
 57:        *Joy1Dat=(USHORT *)0xdff00c;
 58: struct NewWindow nw = {
 59:   0,0,
 60:   WIDTH,HEIGHT,
 61:   0,1,
 62:   IDCMPFLAGS,
 63:   WINDOWFLAGS,
 64:   NULL,NULL,NULL,NULL,NULL,
 65:   WIDTH,HEIGHT,WIDTH,HEIGHT,
 66:   WBENCHSCREEN,
 67: };
 68:
 69: /*
 70:  * Initialisieren der Lichtgriffel Schnittstelle. Leider ist das
 71:  * Amiga-Betriebssystem ein Feind des Lichtgriffels, denn die
 72:  * Copperliste der Intuition-Screens setzt das sogenannte LPEN-Bit
 73:  * ständig zurück. Da diese Copperliste sehr vielfältig
 74:  * aussehen kann, muß sie Befehl für Befehl nach dem verflixten
 75:  * Rücksetzbefehl - oder auch mehreren - durchsucht werden, die
 76:  * dann gepatcht werden. Der Bildschirm wird gelöscht und oben
 77:  * links wird eine kleine Palette gezeichnet.
 78:  */
 79: void InitLPen(struct View *View) {
 80: register USHORT i,*w;
 81:   /* Zum Betrieb des Lichtgriffels muß Bit 3 LPEN von BPLCON0
 82:    * gesetzt werden. Weil BPLCON0 aber in der Copperliste
 83:    * verwendet wird, muß die Copperliste gepatcht werden. Gesucht
 84:    * wird der Befehl MOVE BPLCON0,#xx
 85:    */
 86:
 87:   if(View->LOFCprList) {
 88:     w=View->LOFCprList->start;   /* w zeigt auf den Beginn der
 89:                                     Copperliste */
 90:     /* alle Befehle durchsuchen */
 91:     for(i=0;i<View->LOFCprList->MaxCount;i++,w+=2) {
 92:       if(*w==0xffff && *(w+1)==0xfffe)
 93:          break;         /* WAIT ffff beendet die Copperliste */
 94:       if(*w==0x0100) *(w+1)|=1<<3;
 95:                    /* Befehl gefunden LPEN-Bit setzen (Bit 3) */
 96:     }
 97:   }
 98:   /* ab jetzt ist Light-Pen Modus aktiviert. Zum Auswählen der
 99:    * Farben werden vier Gadgets gezeichnet
100:    */
101:   SetDrMd(LPenPort,JAM1);
102:   SetRast(LPenPort,0);  /* Hintergrund löschen */
103:   for(i=0;i<4;i++)  {
104:     SetAPen(LPenPort,i);  /* Lösch und Farbrechtecke zeichnen */
105:     RectFill(LPenPort,i*20,0,i*20+20,10);
106:     SetAPen(LPenPort,0);  /* ausgefüllt mit Farbe 0 */
```

```
107:      RectFill(LPenPort,i*20+5,3,i*20+15,7);
108:    }
109:    SetAPen(LPenPort,1);
110: }
111:
112: /*
113:  * Abfrage der Lichtgriffelposition. LPen liefert einen Code
114:  * zurück, der angibt, ob der Strahl erkannt wurde und ob
115:  * zusätzlich die Schreibspitze des Griffels betätigt ist.
116:  * Code=0: Lichtgriffel nicht erkannt, zu weit vom Bildschirm
117:  * entfernt
118:  * BEAMDETECT: Lichtgriffelposition erkannt, Position steht in PenX
119:  * und PenY
120:  * PENDOWN: Lichtgriffelspitze ist ausgelöst, das Zeichnen kann
121:  * stattfinden
122:  * PENPRESSED : wird nur im Moment des Aufdrückens zusammen mit
123:  * PENDOWN zurückgegeben
124:  */
125: USHORT LPen(struct View *View) {
126: register USHORT h1,h2,vpos,Code,Pen;
127: register ULONG pos;
128:
129:    h1=custom.vhposr; /* Beamposition lesen */
130:    h2=custom.vhposr; /* nochmal lesen */
131:
132:    if(h1!=h2)         /* ungleich ? */
133:       return(FALSE); /* wenn Register zählt, hat Lichtgriffel nicht
134:                         ausgelöst */
135:    vpos=custom.vposr & 0x0001; /* V8 lesen */
136:
137:    /* Horizontal-Überlauf abfangen, da Werte nur bis 0xe3 gehen */
138:    if((h2=h1&0xff) < 0x20)
139:       /* horizontal < 0x20 ist in Wirklichkeit rechter Bildrand */
140:       h1=(h1 & 0xff00) | (h2+0xe4);
141:
142:       /* kombinierte vertikal Horizontalposition  V8-V0 H8-H1 */
143:    pos=(ULONG)vpos<<16 | h1;
144:
145:    if(pos>(ULONG)(View->ViewPort->DHeight+View->DyOffset)<<8)
146:      /* Vertikalposition ist zu groß, unterer Rand */
147:      return(FALSE);
148:
149:      /* wenn bis hier alles stimmt, ist BEAMDETECT erkannt */
150:    Code=BEAMDETECT;
151:
152:      /* X aus Strahlposition berechnen */
153:    PenX=((pos & 0x00ff)<<1)-View->DxOffset-DXOFFSET;
154:
155:      /* bei HIRES verdoppeln */
156:    if(View->ViewPort->Modes & HIRES) PenX*=2;
157:
158:      /* Y aus Strahlposition errechnen */
159:    PenY=((pos & 0x1ff00)>>8)-View->DyOffset;
160:
161:    Pen=*Joy1Dat & PENDOWN;  /* Schreibknopf gedrückt? */
162:    if(Pen==PENDOWN) {
163:      Code|=PENDOWN;          /* wenn ja setze PENDOWN Flag */
164:      if(OldPen!=PENDOWN)  /* wenn vorher noch nicht gedrückt */
165:        Code |=PENPRESSED; /* setze zusätzlich PENPRESSED Flag */
166:    }
167:    OldPen=Pen;                /* für nächste Abfrage abspeichern */
168:    return(Code);    /* aufrufendes Programm bekommt die Flags */
169: }
170:
171: /*
172:  * Zeichnet ein kleines Fadenkreuz auf den Bildschirm. Es soll
173:  * verdeutlichen, daß nicht erst beim Auftippen der
174:  * Lichtgriffelspitze, sondern schon bei der Annäherung
175:  * an den Bildschirm die Position festgestellt werden kann.
176:  */
177: DrawCrossHair(SHORT x,SHORT y) {
178:    SetDrMd(LPenPort,COMPLEMENT);
179:    Move(LPenPort,x-10,y);
180:    Draw(LPenPort,x+10,y);
181:    Move(LPenPort,x,y-5);
182:    Draw(LPenPort,x,y+5);
183: }
184:
185: /*
186:  * Programm Main() initialisiert den Lichtgriffel und fragt den
187:  * Zustand zyklisch ab. Der Befehl Draw könnte auch durch einen
188:  * RectFill()- oder DrawCircle()- oder BltBitMap()-Aufruf zum
189:  * Kopieren aus einer BitMap ersetzt werden. Dadurch
190:  * ließen sich bei jedem Auftippen Objekte statt Linien zeichnen.
191:  * Oben links befindet sich eine Farbpalette, aus der man sich
192:  * verschiedene Farben durch Auftippen auswählen kann. Die
193:  * Schleife wird durch Drücken der linken Maustaste beendet
194:  */
195: main() {
196:
197: SHORT Code,i;
198: BOOL  CrossHair=FALSE;
199:
200:   /* Öffnen der Libraries */
201:   IntuitionBase= (struct IntuitionBase *)
202:                    OpenLibrary("intuition.library",NULL);
203:   GfxBase =(struct GfxBase *)OpenLibrary("graphics.library",NULL);
204:
205:   if(GfxBase==0 || IntuitionBase==0)
206:      exit(10);
207:
208:   if(!(LPenWindow=OpenWindow(&nw)))  /* eigenes Fenster öffnen */
209:     exit(10);
210:
211:   LPenView=GfxBase->ActiView;
212:   LPenPort=LPenWindow->RPort;
213:   InitLPen(LPenView);          /* Lichtgriffel aktivieren */
214:
215:   do {
216:     Code=LPen(LPenView);       /* Lichtgriffelzustand abfragen */
217:     if(Code & PENPRESSED) {
218:       /* LPen liefert in den globalen Variablen PenX PenY die
219:        * Position des Lichtgriffels, falls er sich oben links
220:        * befindet können Farben gewählt oder der Bildschirm
221:        * gelöscht werden
222:        */
223:
224:      if(CrossHair) DrawCrossHair(OldPenX,OldPenY);
225:      CrossHair=FALSE;   /* Fadenkreuz löschen */
226:
227:      if(PenY<10 && PenX<80) {       /* Farbwahl oder Löschen ? */
228:         if(PenX<20)
229:           InitLPen(LPenView);         /* Bildschirm Löschen! */
230:         else
231:           SetAPen(LPenPort,PenX/20);  /* Farbe wählen */
232:         while(*Joy1Dat & PENDOWN);    /* Griffel abgehoben? */
233:         Code=0;
234:       } else
235:         /* immer wenn der Stift aufgesetzt wird, beginnt eine neue
236:          * Linie */
237:         Move(LPenPort,PenX,PenY);
238:     }
239:
240:     /* Eine Linie wird gezeichnet, wenn der Griffel aufliegt und
241:      * sich seine Position verändert hat, ansonsten ist bei
242:      * Annäherung des Lichtgriffels ein Fadenkreuz zu sehen
243:      */
244:     if(Code & BEAMDETECT && !(Code & PENDOWN)) {
245:       if(CrossHair)
246:         DrawCrossHair(OldPenX,OldPenY);
247:       DrawCrossHair(PenX,PenY);
248:       CrossHair=TRUE;
249:     } else if(Code & BEAMDETECT && Code & PENDOWN &&
250:                  (PenX!=OldPenX || PenY!=OldPenY)) {
251:       SetDrMd(LPenPort,JAM1);
252:       Draw(LPenPort,PenX,PenY);
253:     }
254:     OldPenX=PenX;
255:     OldPenY=PenY;  /* alte Position abspeichern */
256:   }
257:     /* Linke Maustaste beendet das Programm */
258:   while(*(UBYTE *)0xbfe001 & LEFT_BUTTON);
259:
260:   CloseWindow(LPenWindow);
261:   CloseLibrary(IntuitionBase);
262:   CloseLibrary(GfxBase);
263:
264:   /* Die Copperliste wird automatisch korrigiert, wenn man einen
265:    * Screen herunterzieht
266:    */
267: }                                                    © 1993 M&T
```

»LPen.c«: Das Programm demonstriert den Umgang mit einem Light-Pen am Amiga. Sie finden es auch auf der Diskette zum Heft (Seite 114).

Grundlagen Compiler-Bau

Der Mini-Compiler

Compiler-Bau ist eines der interessantesten Gebiete der Informatik. Im folgenden finden Sie daher das erforderliche Grundwissen über die Funktionsweise und den Aufbau eines Compilers.

von Fridtjof Siebert

Aufgabe eines Compilers ist es, einen in einer abstrakten Programmiersprache geschriebenen Quelltext in ein gleichwertiges Programm einer anderen, meist simpleren, Zielsprache zu übersetzen. Ein Compiler besteht gewöhnlich aus drei Modulen: dem Scanner, dem Parser und dem Codegenerator.

❐ Aufgabe des Scanners ist es, den Text des Quellprogramms, der im allgemeinen als ASCII-Datei vorliegt, in eine Folge von Symbolen der Quellsprache zu wandeln. Diese Symbole lassen sich später leichter verarbeiten als einzelne Zeichen des Textes.

❐ Der Parser erhält die Symbole als Eingabe. Er interpretiert die Symbolfolge und prüft, ob das Quellprogramm der Syntax der Quellsprache entspricht. Dabei findet der Parser auch heraus, aus welchen Elementen der Syntax das Programm besteht.

❐ Der letzte Programmteil, der Codegenerator, erzeugt aus den vom Parser erhaltenen Informationen ein gleichwertiges Programm der Zielsprache.

Program	PROGRAM [VarDeclaration] BEGIN {Statement} END .
VarDeclaration	VAR Identifier{"," Identifier}.
Statement	Assignment | While | Print .
Assignment	Identifer "=" Expression .
While	WHILE Expression DO { Statement } END .
Print	PRINT Expression
Expression	["+" | "-"] Factor { ("+" | "-") Factor } .
Factor	Identifier | Constant .
Identifier	Letter { Letter } .
Constant	Digit { Digit } .
Letter	"a" | .. | "z" | "A" | .. | "Z" .
Digit	"0" | .. | "9" .

Die Syntax der Miniprogrammiersprache in EBNF-Notation

Mini

Betrachten wir als Beispiel für eine Quellsprache die Sprache »Mini«. Die Abbildung zeigt ihre Syntax. Zur Beschreibung der Syntax wird die weit verbreitete »erweiterte Backus-Naur-Form« (EBNF, [1]) verwendet. Ähnlichkeiten mit Modula-2 sind unverkennbar, allerdings ist sie viel einfacher. Alle Variablen sind automatisch vom Typ »INTEGER«. Der Typ wird bei der Deklaration daher nicht angegeben. Als Anweisungen kennt Mini nur die Zuweisung, eine Schleife und eine Ausgabefunktion. Die Schleife wird wie in Modula-2 mit dem Schlüsselwort »WHILE« eingeleitet. Sie wird solange durchlaufen, bis der Ausdruck hinter WHILE den Wert 0 annimmt.

Bei Ausdrücken erlaubt Mini nur das Rechnen mit den Operatoren »+« und »-«. Trotz der Simplizität lassen sich mit Mini kleinere Programme schreiben. Das Miniprogramm »Fibonacci.mini« (Listing 1) errechnet die ersten 46 Fibonacci-Zahlen und gibt sie auf dem Bildschirm aus (eine Fibonacci-Zahl ist die Summe der beiden vorhergehenden Fibonacci-Zahlen, wobei die ersten zwei den Wert 1 besitzen).

```
 1: PROGRAM
 2:
 3: VAR
 4:   a,b,c
 5:
 6: BEGIN
 7:   a = 1
 8:   b = 0
 9:   c = 46
10:   WHILE c DO
11:     PRINT a
12:     a = a + b
13:     b = a - b
14:     c = c - 1
15:   END
16: END
17:
18:             © 1993 M&T
```

Listing 1. Unser »Mini«-Programm berechnet die ersten 46 Fibonacci-Zahlen

Jetzt geht es darum, einen Mini-Compiler zu schreiben. Als Sprache verwenden wir Amiga-Oberon. Der Compiler läßt sich mit Amiga-Oberon 2.14/3.0 oder der Demoversion von Amiga-Oberon 3.0 [2] übersetzen. Die Demoversion finden Sie auch auf der Diskette zum Heft (siehe Seite 114).

Der Scanner

Zunächst benötigen wir den »Mini-Scanner« (Listing 2). Der Scanner soll den Quelltext, der als gewöhnliche Textdatei vorliegt, in eine Symbolfolge zerlegen. Für jedes Symbol wird dazu eine (INTEGER-)Konstante definiert. Die Symbole sind die reservierten Wörter, spezielle Zeichen (Komma, Gleichheitszeichen usw.), Bezeichnernamen und Konstanten. Das spezielle Symbol »eof« kennzeichnet das Ende der Quelltextdatei.

Der Scanner exportiert die Prozedur »GetSym«, die das jeweils nächste Symbol des Quelltextes bestimmt und die Variable sym auf die für dieses Symbol vorgesehene Konstante setzt. Ist das Symbol ein Bezeichner (»ident«), wird zusätzlich der Bezeichnername in der Variablen »identifier« abgelegt. Ist es eine Konstante (»const«), wird ihr Wert in »constant« gesichert.

Von der Prozedur »GetChar« erhält GetSym nacheinander die Zeichen des Quelltextes. Mehrere aufeinanderfolgende Buchstaben werden zusammengefaßt und mit den reservierten Wörtern verglichen. Bilden sie ein reserviertes Wort, wird die Konstante des entsprechenden Symbols, ansonsten ident in die Variable sym geschrieben. Trifft der Scanner auf Ziffern, berechnet er den Wert

der so gebildeten Dezimalzahl und speichert ihn in der Variablen constant: sym wird auf const gesetzt. Bei speziellen Zeichen wird sym auf die für diese Symbole definierte Konstante gesetzt. Ein Zeichen, das keinem Symbol der Sprache Mini entspricht, führt zu einer entsprechenden Fehlermeldung und zum Programmabbruch.

```
 1: MODULE Scanner;
 2: IMPORT F := FileSystem, R := Requests;
 3:
 4: CONST     (* Symbole von Mini *)
 5:  program * =0; var   * = 1; begin * = 2;
 6:  end     * =3; while * = 4; do    * = 5;
 7:  print   * =6; comma * = 7; equal * = 8;
 8:  plus    * =9; minus * =10; ident * = 11;
 9:  const   * =12; eof   * = 13;
10:  MaxIdLen * = 79;
11:
12: TYPE Identifier * = ARRAY MaxIdLen+1 OF C
    HAR;
13:
14: VAR
15:  source  * : F.File; (* Quelltextdatei*)
16:  char      : CHAR;      (* zuletzt gel
    esenes Zeichen (0X bei EOF) *)
17:  sym       - : INTEGER;    (* das letzte
    Symbol (program, var, etc.) *)
18:  identifier- : Identifier; (* der Bezeich
    ner bei sym=ident           *)
19:  constant  - : LONGINT;    (* die Konstan
    te bei sym=const            *)
20:
21: PROCEDURE GetChar;
22: BEGIN IF ~ F.Read(source,char) THEN char
    := 0X END END GetChar;
23:
24: PROCEDURE GetSym*;
25: VAR i: INTEGER;
26: BEGIN
27:  WHILE (char>0X) & (char<=" ") DO GetChar
    END;   (* Space, LF, etc. *)
28:  CASE CAP(char) OF
29:  | "A".."Z":      (* Bezeichner oder Schlü
    sselwort *)
30:     i := 0;
31:     REPEAT
32:      identifier[i] := char;
33:      INC(i);
34:      GetChar;
35:     UNTIL (i=MaxIdLen) OR (CAP(char)<"A")
    OR (CAP(char)>"Z");
36:     identifier[i] := 0X;
37:     IF    identifier = "PROGRAM" THEN sym
    := program
38:     ELSIF identifier = "VAR"     THEN sym
    := var
39:     ELSIF identifier = "BEGIN"   THEN sym
    := begin
40:     ELSIF identifier = "END"     THEN sym
    := end
41:     ELSIF identifier = "WHILE"   THEN sym
    := while
42:     ELSIF identifier = "DO"      THEN sym
    := do
43:     ELSIF identifier = "PRINT"   THEN sym
    := print
44:     ELSE
45:      sym := ident;         (* Kein Schlüs
    selwort, also Bezeichner *)
46:     END;
47:  | "0".."9":
               (* Konstante Zahl *)
48:     constant := 0;
49:     REPEAT
50:      constant := 10*constant + ORD(char) -
      ORD("0"); GetChar;
51:     UNTIL (char<"0") OR (char>"9");
52:     sym := const;
53:  | "=": sym := equal; GetChar;
               (* Sonderzeichen *)
54:  | ",": sym := comma; GetChar;
55:  | "+": sym := plus ; GetChar;
56:  | "-": sym := minus; GetChar;
57:  | 0X : sym := eof;
58:  ELSE R.Fail("Unerwartetes Zeichen!") END
   ;
59: END GetSym;
60:
61: BEGIN char := " " END Scanner. © 1993 M&T
```

Listing 2: Das Scanner-Modul überprüft den Quelltext Bezeichnern

Der Parser

Nun gilt es, den Parser zu entwerfen. Bei vielen Sprachen – vor allem bei den Wirth´schen – bietet sich ein Parser an, der nach dem Prinzip des rekursiven Abstiegs arbeitet [1]. Auch für Mini wählen wir diese Methode. Für komplexere Sprachen gibt's aufwendigere Algorithmen; sie werden in [3] ausführlich beschrieben. Listing 3 enthält den Quelltext des Mini-Parsers. Die Aufrufe der Routinen aus dem Modul Code sind nicht Teil des Parsers. Wir gehen später darauf ein.

Ein mit rekursivem Abstieg arbeitender Parser enthält für die Symbole aus der EBNF-Syntax der untersuchten Sprache jeweils eine Prozedur. Diese Prozeduren haben die Aufgabe, das entsprechende Sprachkonstrukt zu untersuchen. Aus der in EBNF gegebenen Syntax-Beschreibung läßt sich ohne Umwege direkt ein rekursiver Abstiegs-Parser erzeugen.

Es ist zweckmäßig, mit dem »oberen« Symbol der Grammatik zu beginnen – bei Mini ist es das Symbol »Program«. Die gleichnamige Prozedur des Parsers wertet dieses Symbol aus. Zunächst überprüft sie, ob der Quelltext mit dem Schlüsselwort »PROGRAM« beginnt. Da an verschiedenen Stellen im Parser ein bestimmtes Symbol zu checken ist, wurde hierfür die Prozedur »Chk« implementiert. Chk liest auch gleich das folgende Symbol ein.

```
 1: MODULE Parser;
 2:
 3: IMPORT S := Scanner, R := Requests, Code;
 4:
 5: PROCEDURE Chk(sym: INTEGER; msg: ARRAY OF
    CHAR);
 6: BEGIN IF S.sym=sym THEN S.GetSym ELSE R.F
    ail(msg) END END Chk;
 7:
 8: PROCEDURE VarDeclaration;
 9: (* VarDeclaration -> VAR Identifier {","
    Identifier} . *)
10: BEGIN
11:   REPEAT
12:     S.GetSym;
13:     IF S.sym#S.ident THEN R.Fail("Bezeich
    ner in erwartet!") END;
14:     Code.DCL(S.identifier);
15:     S.GetSym;
16:   UNTIL S.sym#S.comma;
17: END VarDeclaration;
18:
19: PROCEDURE Factor;
20: (* Factor        -> Identifier | Constan
    t . *)
21: BEGIN
22:   CASE S.sym OF
23:   | S.ident : Code.MOVEvarD0  (S.identifi
    er); S.GetSym;
24:   | S.const : Code.MOVEconstD0(S.constant
      ); S.GetSym;
25:   ELSE R.Fail("Faktor erwartet") END;
26: END Factor;
27:
28: PROCEDURE Expression;
29: (* Expression     -> [ "+" | "-" ] Factor
    { ( "+" | "-" ) Factor } . *)
30: VAR neg: BOOLEAN;   (* muß D0 negiert wer
    den? *)
31: BEGIN
32:   neg := FALSE;
33:   IF S.sym IN {S.plus,S.minus} THEN neg :
    = S.sym=S.minus; S.GetSym END;
34:   Factor;
35:   IF neg THEN Code.NEGD0 END;
36:   WHILE S.sym IN {S.plus,S.minus} DO
37:     neg := S.sym=S.minus; S.GetSym;
38:     Code.MOVED0D1;
39:     Factor;
40:     IF neg THEN Code.NEGD0 END;
41:     Code.ADDD1D0;
42:   END;
43: END Expression;
44:
45: PROCEDURE Statement;
46: (* Statement      -> Assignment | While |
    Print .              *)
47: (* Assignment     -> Identifer "=" Expres
    sion .               *)
48: (* While          -> WHILE Expression DO
    { Statement } END . *)
49: (* Print          -> PRINT Expression
                         *)
50: VAR
51:   varname: S.Identifier;
       (* Ziel einer Zuweisung *)
52:   start,end : INTEGER;
    (* Labels bei einer Schleife *)
53: BEGIN
54:   CASE S.sym OF
55:   | S.ident: varname := S.identifier; S.G
    etSym;            (* Zuweisung *)
56:              Chk(S.equal,"'=' erwartet!")
    ;
57:              Expression;
58:              Code.MOVED0var(varname);
59:   | S.while: S.GetSym;
                 (* Schleife *)
60:              start := Code.GetLabel(); en
    d := Code.GetLabel();
61:              Code.Label(start);
62:              Expression;
63:              Chk(S.do,"DO erwartet!");
64:              Code.TSTD0; Code.BLE(end);
65:              WHILE S.sym#S.end DO Stateme
    nt END;
66:              Chk(S.end,"END erwartet!");
67:              Code.BRA(start); Code.Label(
    end);
68:   | S.print: S.GetSym; Expression; Code.P
    rintD0;            (* Ausgabe *)
69:   ELSE R.Fail("Statement erwartet!") END;
70: END Statement;
71:
72: PROCEDURE Program*;
73: (* Program -> PROGRAM [ VarDeclaration ]
    BEGIN { Statement } END. *)
74: VAR start: INTEGER;
75: BEGIN
76:   S.GetSym; Chk(S.program,"PROGRAM erwart
    et!");
77:   start := Code.GetLabel(); Code.BRA(star
    t);
78:   IF S.sym=S.var THEN VarDeclaration END;
79:   Chk(S.begin,"BEGIN erwartet!");
80:   Code.StartUp(start);
81:   WHILE S.sym#S.end DO Statement END;
82:   Code.CleanUp;
83:   Chk(S.end,"END erwartet!");
84: END Program;
85:
86: END Parser.   © 1993 M&T
```

Listing 3: Dem Parser-Modul checkt das Programm auf korrekte Synta

```
 1: MODULE Code;
 2:
 3: IMPORT io;
 4:
 5: VAR labels: INTEGER;
 6:
 7: PROCEDURE W (s: ARRAY OF CHAR); BEGIN io.
    WriteString(s) END W;
 8: PROCEDURE WL(s: ARRAY OF CHAR); BEGIN W(s
    ); io.WriteLn  END WL;
 9:
10: PROCEDURE DCL*(var: ARRAY OF CHAR);    (*
    Speicher für var reservieren *)
11: BEGIN W(var); WL(": dc.l 0") END DCL;
12:
13: PROCEDURE MOVEvarD0*(var: ARRAY OF CHAR);
          (* var nach D0 kopieren *)
14: BEGIN W("    MOVE.L  "); W(var); WL(",D0"
    ) END MOVEvarD0;
15:
16: PROCEDURE MOVEconstD0*(c: LONGINT);
    (* Konstante c nach D0 laden *)
17: BEGIN W("    MOVE.L  #"); io.WriteInt(c,0
    ); WL(",D0") END MOVEconstD0;
18:
19: PROCEDURE MOVED0var*(name: ARRAY OF CHAR)
    ;        (* D0 in D0 kopieren *)
20: BEGIN W("    MOVE.L  D0,"); WL(name) END
    MOVED0var;
21:
22: PROCEDURE MOVED0D1*; BEGIN WL("    MOVE.L
      D0,D1") END MOVED0D1;
23: PROCEDURE ADDD1D0 *; BEGIN WL("    ADD.L
      D1,D0") END ADDD1D0;
24: PROCEDURE NEGD0   *; BEGIN WL("    NEG.L
      D0"   ) END NEGD0;
25: PROCEDURE TSTD0   *; BEGIN WL("    TST.L
      D0"   ) END TSTD0;
26:
27: PROCEDURE GetLabel*(): INTEGER;
         (* neues Label anfordern *)
28: BEGIN INC(labels); RETURN labels END GetL
    abel;
29:
30: PROCEDURE Label*(l: INTEGER);
              (* Label <l> ausgeben *)
31: BEGIN W("L"); io.WriteHex(l,3); WL(":") E
    ND Label;
32:
33: PROCEDURE BLE*(l: INTEGER);
              (* BLE <l> erzeugen *)
34: BEGIN W("    BLE     L"); io.WriteHex(l,3
    ); WL("") END BLE;
35: PROCEDURE BRA*(l: INTEGER);
              (* BRA <l> erzeugen *)
36: BEGIN W("    BRA     L"); io.WriteHex(l,3
    ); WL("") END BRA;
37:
38: PROCEDURE PrintD0*;
         (* Wert von D0 ausgeben *)
39: BEGIN
40:   WL("    LEA     _format,A0");
41:   WL("    MOVE.L  A0,D1"     );
42:   WL("    LEA     _print,A0" );
43:   WL("    MOVE.L  A0,D2"     );
44:   WL("    MOVE.L  D0,(A0)"   );
45:   WL("    MOVE.L  _dos,A6"   );
46:   WL("    JSR     -954(A6)"  );
47: END PrintD0;
48:
49: PROCEDURE StartUp*(start: INTEGER);
    (* Dos-Library öffnen, etc. *)
50: BEGIN
51:   WL("_dos:      DC.L   0"              );
52:   WL("_dosname:  DC.B   'dos.library',0");
53:   WL("_format:   DC.B   '%ld',10,0"     );
54:   WL("           DS.L   0"              );
55:   WL("_print:    DC.L   0"              );
56:   Label(start);
57:   WL("    LEA     _dosname,A1"          );
58:   WL("    MOVE.L  #37,D0"               );
59:   WL("    MOVE.L  $4,A6"                );
60:   WL("    JSR     -552(A6)"             );
61:   WL("    TST.L   D0"                   );
62:   WL("    BNE.S   _ok"                  );
63:   WL("    RTS"                          );
64:   WL("_ok:"                             );
65:   WL("    MOVE.L  D0,_dos"              );
66: END StartUp;
67:
68: PROCEDURE CleanUp*;                    (*
    Dos-Library schließen, etc. *)
69: BEGIN
70:   WL("    MOVE.L  _dos,A1" );
71:   WL("    MOVE.L  $4,A6"   );
72:   WL("    JSR     -414(A6)");
73:   WL("    MOVE.L  #0,D0"   );
74:   WL("    RTS"             );
75:   WL("    END"             );
76: END CleanUp;
77:
78: BEGIN labels := 0 END Code.  © 1993 M&T
```

Listing 4: Das Code-Modul generiert den adäquaten Assembler-Code.

Der optionale Variablendeklarationsteil wird im Parser mit einer IF-Anweisung ausgewertet: Trifft der Parser auf das Symbol »VAR«, springt der die Prozedur »VarDeclaration« an. Im Deklarationsteil dürfen sich beliebig viele, mindestens jedoch eine, Variablendeklaration befinden. Daraus folgt, daß die Untersuchung am einfachsten mit einer REPEAT-Schleife durchzuführen ist.

Die dem Schlüsselwort »BEGIN« folgenden »Statements«, deren Anzahl im übrigen unbegrenzt sein darf, wertet man am besten mit einer WHILE-Schleife aus. Ein Statement setzt sich aus drei Alternativen zusammen: »Assignment«, »While« und »Print«. Eine Case-Anweisung zur Auswertung erscheint hier am sinnvollsten. Um nicht im Wust zuvieler Prozeduren die Übersicht zu verlieren, belassen wir es in diesem Fall bei einer: der Prozedur »Statement«. Diese wertet alle Anweisungen direkt aus.

```
 1: MODULE Mini;
 2:
 3: IMPORT Parser, Scanner, F := FileSystem,
    R := Requests, Arguments;
 4:
 5: VAR name: ARRAY 80 OF CHAR;
 6:
 7: BEGIN
 8:   R.Assert(Arguments.NumArgs()=1,"Aufruf:
    Mini <Quelltext>");
 9:   Arguments.GetArg(1,name);
10:   R.Assert(F.Open(Scanner.source,name,FAL
    SE),"Text nicht gefunden!");
11:   Parser.Program;
12:   IF F.Close(Scanner.source) THEN END;
13: END Mini.     © 1993 M&T
```

Listing 5: Im Hauptmodul laufen alle Fäden zusammen.

Beim Abarbeiten der WHILE-Schleife wird deutlich, weshalb die Parsing-Methode »rekursiver Abstieg« heißt: Um die Anweisungen im Schleifenrumpf zu untersuchen, ruft man die Prozedur Statement rekursiv auf.

Ausdrücke und Faktoren werden von den Prozeduren »Expression« bzw. »Factor« geprüft. Im Vergleich zu den bisher besprochenen Funktionen enthalten sie jedoch nichts entscheidend Neues. Um schnell festzustellen, ob das aktuelle Symbol einer der Operatoren + oder - ist, wird dieses mit der Menge »{S.plus,S.minus}« verglichen.

Der Codegenerator

In den bisherigen Teilen sind sich viele Compiler sehr ähnlich. Die wirklich interessanten Aufgaben kommen erst jetzt auf uns zu. Der Codegenerator formuliert die Bedeutung des Quellprogramms in der Zielsprache. Die Effizienz des übersetzten Programms hängt entscheidend von der Qualität des Codegenerators ab. Als Zielsprache wählen wir 68000-Assembler-Quelltext, der mit auf dem Amiga verbreiteten Assemblern in Maschinencode zu übersetzen ist. Geeignet ist beispielsweise der frei kopierbare Assembler »a68k« von Charlie Gibb [4].

```
 1:      BRA     L001
 2: a: dc.l 0
 3: b: dc.l 0
 4: c: dc.l 0
 5: _dos:       DC.L    0
 6: _dosname:   DC.B    'dos.library',0
 7: _format:    DC.B    '%ld',10,0
 8:             DS.L    0
 9: _print:     DC.L    0
10: L001:
11:      LEA     _dosname,A1
12:      MOVE.L  #37,D0
13:      MOVE.L  $4,A6
14:      JSR     -552(A6)
15:      TST.L   D0
16:      BNE.S   _ok
17:      RTS
18: _ok:
19:      MOVE.L  D0,_dos
20:      MOVE.L  #1,D0
21:      MOVE.L  D0,a
22:      MOVE.L  #0,D0
23:      MOVE.L  D0,b
24:      MOVE.L  #46,D0
25:      MOVE.L  D0,c
26: L002:
27:      MOVE.L  c,D0
28:      TST.L   D0
29:      BLE     L003
30:      MOVE.L  a,D0
31:      LEA     _format,A0
32:      MOVE.L  A0,D1
33:      LEA     _print,A0
34:      MOVE.L  A0,D2
35:      MOVE.L  D0,(A0)
36:      MOVE.L  _dos,A6
37:      JSR     -954(A6)
38:      MOVE.L  a,D0
39:      MOVE.L  D0,D1
40:      MOVE.L  b,D0
41:      ADD.L   D1,D0
42:      MOVE.L  D0,a
43:      MOVE.L  a,D0
44:      MOVE.L  D0,D1
45:      MOVE.L  b,D0
46:      NEG.L   D0
47:      ADD.L   D1,D0
48:      MOVE.L  D0,b
49:      MOVE.L  c,D0
50:      MOVE.L  D0,D1
51:      MOVE.L  #1,D0
52:      NEG.L   D0
53:      ADD.L   D1,D0
54:      MOVE.L  D0,c
55:      BRA     L002
56: L003:
57:      MOVE.L  _dos,A1
58:      MOVE.L  $4,A6
59:      JSR     -414(A6)
60:      MOVE.L  #0,D0
61:      RTS
62:      END     © 1993 M&T
```

Listing 6. Der von Mini erzeugte Assembler-Code für das Fibonacci-Programm

Die meisten Compiler erzeugen direkt Maschinencode und speichern ihn als Objektdatei oder als ausführbares Programm. Wir legen keinen besonderen Wert auf die Effizienz, dennoch ist der erzeugte Code akzeptabel. Für die Erzeugung der Anweisungen der Zielsprache ist es zweckmäßig, spezielle Funktionen zu schreiben. In unserem Beispiel sind sie im Modul »Code« (Listing 4) zusammengefaßt. Das erzeugte Assembler-Programm wird mit den Prozeduren des Moduls »io« einfach auf dem Bildschirm ausgegeben.

»DCL« generiert die Assembler-Anweisung »dc.l 0«. Sie reserviert Speicher für eine Variable. Die Adresse der Variablen wird mit einem Label (Markierung) versehen. Dabei verwenden wir den Variablennamen.

Die Befehle »MOVEvarD0«, »MOVEconstD0«, »MOVED0var« und »MOVED0D1« bilden die entsprechenden Move-Befehle mit Variablen, Registern oder Konstanten als Argument. »ADDD1D0«, »NEGD0« und »TSTD0« entsprechen den gleichnamigen Assembler-Befehlen für die Prozessorregister D1 und D0.

Für Sprunganweisungen im Zielprogramm benötigen wir weitere Label. Diese werden auch vom Modul Code verwaltet. Dabei ist jedes Label vor der Verwendung mit »GetLabel« anzufordern. Das Ergebnis ist ein der Markierung eindeutig zugeordneter Integer-Wert. Die Prozedur »Label« gibt die übergebene Marke an der aktuellen Position im Zielprogramm aus. »BLE« und »BRA« erzeugen die entsprechenden bedingten bzw. unbedingten Verzweigungsbefehle mit dem übergebenen Label als Ziel.

Anders als die bisherigen Prozeduren des Codemoduls erzeugt »Print« eine ganze Palette Assembler-Befehle, die den Wert des Registers D0 mit der Routine »VPrintf« der DOS-Library (V37) ausgeben [5]. Die von unserem Mini-Compiler erzeugten Programme sind also nur ab Betriebssystem 2.0 oder höher lauffähig.

»Single Pass«: Übersetzung in einem Durchlauf

»StartUp« und »CleanUp« erzeugen den Code, der zu Beginn bzw. am Schluß des Programms ausgeführt werden muß. Hier findet man einige Variablen- und Konstantendefinitionen. Weiterhin wird die DOS-Library geöffnet bzw. geschlossen. Sollen die Programme auch von der Workbench lauffähig sein, ist der Startup-Code entsprechend zu erweitern [6].

Ein effizienter Compiler arbeitet meist nach dem sog. »single pass«-Prinzip. Dabei wird in einem einzigen Durchlauf der Quelltext untersucht und das Zielprogramm erzeugt. Um das zu erreichen, erweitert man den Parser, um die für die Codeerzeugung nötigen Anweisungen. Bei unserem Mini-Compiler sind dies die Aufrufe der Routinen des Codemoduls im Modul Parse (Listing 3). Der erste Befehl erzeugt in der Prozedur »Program« einen Sprung an den Anfang des Startup-Codes. Zudem wurde »Program« um die Aufrufe von »Code.StartUp« und »Code.CleanUp« erweitert.

In »VarDeclaration« wird mit Code.DCL Speicher für die Variablen reserviert. Für die Auswertung von Faktoren und Ausdrücken wählt man hier ein relativ einfaches Verfahren: Nach einem Aufruf von Factor bzw. Expression befindet sich der Wert des Faktors bzw. des Ausdrucks immer im Register D0. In Expression muß der Wert des ersten Faktors daher in ein anderes Register (hier D1) gerettet werden, bevor Factor erneut aufrufbar ist. Dies geschieht mit dem von Code.MOVED0D1 erzeugten Move-Befehl.

Für eine Zuweisung wird in der Prozedur Statement ein einfacher Move-Befehl erzeugt, der den Wert des in D0 stehenden Ausdrucks in den für die Variable reservierten Speicher schreibt.

Mini schafft sogar Schleifen

Eine Schleife zu übersetzen ist etwas aufwendiger. Hier gibt man zunächst das Label »start« vor der Schleife aus. Dann wird der Schleifenausdruck berechnet und geprüft, ob er kleiner oder gleich 0 ist. Ist das der Fall, springt man ans Label »end« hinter der Schleife. Ansonsten führt man den Schleifenrumpf aus und verzweigt wiederum zur Markierung start. Für eine Print-Anweisung wird die im Modul »Code« programmierte Routine verwendet.

Der Compiler

Es ist soweit. Die drei Bestandteile des Compilers, der Scanner, der Parser und der Codegenerator, sind fertig und lassen sich zu einem Programm zusammenfügen. Mini (Listing 5) ist das Hauptmodul des Compilers. Der an Mini übergebene Quelltext wird in Assembler-Code übersetzt und auf dem Bildschirm ausgegeben. Mit dem Aufruf

```
Mini >Fibonacci.s Fibonacci.mini
```

wird das Beispielprogramm »Fibonacci.mini« (Listing 1) ins Assembler-Programm »Fibonacci.s« (Listing 6) übersetzt.

```
a68k Fibonacci.s
```

assembliert das Programm. Mit dem PD-Linker »BLink« oder dem Oberon-Linker »OLink« läßt sich nun ein ausführbares Programm erzeugen:

```
BLink Fibonacci.o TO Fibonacci
```

oder

```
OLink FROM Fibonacci.o TO Fibonacci
```

Optimierungen

Der bisher beschriebene Mini-Compiler legt keinen Wert auf die Generierung von besonders effizientem Code. Bei kommerziellen Compilern ist das jedoch nötig, um die Programmierung in Assembler überflüssig zu machen.

Einer der wichtigsten Aspekte eines guten Codegenerators ist die Registervergabe und Registerausnutzung. Insofern sollte der Compiler möglichst alle Prozessorregister nutzen und nicht, wie unser Mini-Compiler lediglich zwei. Viele Zugriffe auf den Speicher – und somit auch wertvolle Rechenzeit – läßt sich so einsparen, wenn sich der Compiler merkt, welche Werte oder Variableninhalte in Registern enthalten sind. Dann müssen Variablen nicht ständig neu gelesen werden; es können statt dessen direkt die Werte aus den Registern verwendet werden.

Wichtig wird auch ein Schutzmechanismus für Register, wenn ein Compiler für eine Sprache entwickelt wird, die komplexere Ausdrücke als die von Mini erlaubten ermöglicht. Beim Auswerten eines Teilausdrucks dürfen die Zwischenergebnisse anderer Teilausdrücke, die in Prozessor-Registern gespeichert sind, nicht überschrieben werden.

Eine einfache aber sehr sinnvolle Optimierung ist über einen sog. »Peephole-Optimizer« (Guckloch-Optimierer) möglich. Dabei werden einzelne Assembler-Befehle untersucht, bevor sie ins Zielprogramm geschrieben werden. Können sie durch gleichwertige, schnellere oder kürzere Befehle ersetzt werden, benutzt der Compiler selbstverständlich diese. Typische Beispiele sind etwa die Multiplikation mit einer zweier-Potenz: Sie kann durch eine einfache Schiebeoperation ersetzt werden. Eine Addition der Konstanten 0 wird ignoriert.

Optimierungen: Aus Mini wird Maxi

Bei der Übersetzung geschachtelter Kontrollstrukturen (z.B. »if-then-else«) und Schleifen entstehen oft unnötige Sprunganweisungen, etwa eine bedingte auf eine unbedingte. Diese lassen sich insofern optimieren, daß die bedingte Sprunganweisung gleich ans Ziel der unbedingten springt. Zudem kann hierbei auch gleich nach »totem« Code gesucht werden. Mit totem bezeichnet man Code, der einer unbedingten Sprunganweisung folgt und auf den selbst keine Sprunganweisung existiert. Dieser kann komplett aus dem Zielprogramm entfernt werden.

Es gibt viele Möglichkeiten, den hier beschriebenen Mini-Compiler weiterzuentwickeln. Es ist sogar recht einfach, neue Funktionen (z.B. neue Rechenoperationen) oder fehlende Kontrollstrukturen (z.B. eine if-Anweisung) zu implementieren. Vielleicht kann der Compiler sogar soweit ergänzt werden, daß er eine brauchbare kleine Sprache übersetzt. *rz*

Literatur und Software:

[1] Niklaus Wirth: Compilerbau, BG Teubner Stuttgart, 1986

[2] Fridtjof Siebert: Amiga Oberon 3.0 Demo, AMOK 75

[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilerbau, Addison-Wesley 1988

[4] Charlie Gibb: a68k, Fish 521

[5] Commodore-Amiga Inc.: The AmigaDOS Manual, 3rd Edition, Bantam Books 1991

[6] Commodore-Amiga Inc.: Amiga ROM Kernel Reference Manual Libraries, 3rd Edition, Addison-Wesley 1992

Algorithmen für Strategiespiele

Programmieren mit Taktik

Die Künstliche Intelligenz übertrifft im Bereich der Strategiespiele heute schon die natürliche Intelligenz der meisten Menschen. Wie ist es möglich, daß aus einem Rechenknecht, der im Grunde nur Nullen und Einsen in seinem Speicher verknüpft und verschiebt, ein intelligenter Spielpartner wird? Die grundlegenden Algorithmen sollen hier erläutert werden. Als Beispiel für die Umsetzung in ein MODULA-2-Programm stellen wir Ihnen die Strategieroutinen von »SOGO« vor.

von Bernfried Brüggemann

Die Spielregeln für SOGO sind einfach: In das SOGO-Spielbrett (Bild) werden in einer 4x4-Anordnung insgesamt 16 Stifte aufgestellt. Auf jeden lassen sich je 4 Kugeln auffädeln. Insgesamt sind 64 Kugeln im Spiel, 32 weiße und 32 schwarze. Der Spieler mit den weißen Kugeln beginnt. Beide Spieler setzen abwechselnd eine Kugel ihrer Farbe und versuchen, eine Mühle zu bilden. Eine Mühle besteht aus 4 Kugeln einer Farbe in einer Reihe. Im Spielfeld sind 16 senkrechte, 40 waagerechte, 18 diagonale und 2 raumdiagonale Mühlen möglich. Wer zuerst eine Mühle mit seinen Steinen gebildet hat, gewinnt.

Erst die Theorie...

Bei Strategiespielen wie Schach, Go oder auch SOGO besitzen beide Spielpartner jederzeit alle Informationen über den aktuellen Spielzustand. Es gibt keine verdeckten Karten und auch der Zufall in Gestalt des Würfels spielt keine Rolle. Der Spielablauf ist im Prinzip genau kalkulierbar. Die Spielregeln legen die möglichen Züge in jeder Spielstellung eindeutig fest. Die Spieler können daraus völlig frei den Zug auswählen, den sie für den besten halten.

Strategiespiele dieser Art sind Nullsummenspiele. Bei Spielende entspricht der Gewinn des einen dem Verlust des anderen Spielers. Ein Unentschieden oder Remis ist in diesem Sinne ein Sonderfall.

Der Spielbaum

Auf den ersten Blick scheinen Spiele wie Schach, Go oder SOGO sehr verschieden. Sie haben unterschiedliche Spielbretter, andere Figuren und vor allem verschiedene Spielregeln. Im wesentlichen sind sich alle Strategiespiele dieser Art jedoch sehr ähnlich. Das wird deutlich, wenn man ihren Spielbaum betrachtet. Der Spielbaum gibt einen Überblick aller im Spiel möglichen Stellungen.

Bild 1 zeigt an einem Beispiel die Grundstruktur eines Spielbaums. Die Zweige entsprechen den möglichen Zügen, die Knoten den jeweils erreichten Stellungen. Im Beispiel symbolisiert die Wurzel A des Spielbaums die Anfangsstellung des Spiels. Von dort führen drei Züge zu den Folgestellungen B, F und J. Der Spielbaum mündet schließlich in die Knoten der Endstellungen a-ö des Spiels. Die Spielregeln bestimmen die Bewertung der Knoten als Gewinn oder Verlust. Beim Schach entsprechen Matt-, Remis- oder Pattstellungen den Endknoten.

Die Tiefe einer Stellung im Spielbaum gibt an, wieviel Züge von der Wurzel bis zu dieser Stellung nötig sind. Die Endstellungen in Bild 1 haben somit alle die Tiefe 3.

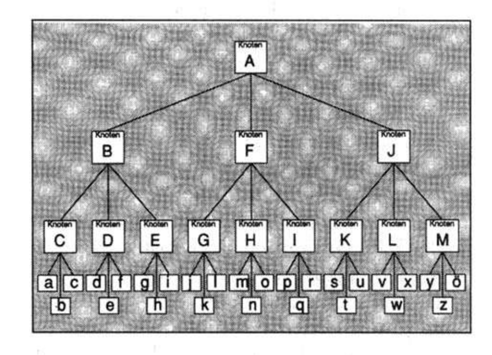


Bild 1: Grundstruktur eines Spielbaums; ob Go, Schach oder SOGO

In der Spielbaumdarstellung unterscheiden sich verschiedene Strategiespiele nur noch in der Zahl der Verästelungen an jedem Knoten, der Zahl der Züge von der Wurzel bis zu einer Endstellung und in der Bewertung dieser Endstellungen. Man sieht dem fertig konstruierten Baum nicht mehr an, welche Spielregeln zugrunde liegen. Sie sind in der Baumstruktur verborgen.

Minimaxverfahren

So einen Spielbaum entwickelt der Computer bei der Suche nach dem besten Zug und testet systematisch alle möglichen Züge. Am Beispiel von Bild 2 auf der folgenden Seite kann man seinen »Gedankengang« verfolgen.

In Stellung A ist der Computer (Weiß) am Zug. Er zieht probeweise nach B und nimmt im ersten Schritt an, daß der Gegenzug von Schwarz zur Stellung C führt. Hier ist der Computer wieder am Zug. Er wählt aus den Endstellungen a, b und c die mit der günstigsten Bewertung, also b. Der Wert 0 dieser Stellung wird für die Stellung C eingetragen.

Der Rechner geht zur Stellung B zurück und zieht im zweiten Schritt nach D. Dort erreicht der Computer mit Stellung f sogar den Wert 8 und trägt ihn für Stellung D ein.

Wieder kehrt der Rechner zur Stellung B, um im dritten Schritt den Gegenzug nach E zu testen. Dort wird der Wert 6 eingetragen.

Damit sind alle möglichen Gegenzüge von Schwarz in Stellung B ausgetestet. Welchen Gegenzug wird Schwarz wählen? Schwarz

zieht so, daß der Computer (Weiß) anschließend nur noch möglichst schlecht bewertete Endstellungen erreichen kann und wählt daher die Position C. Der Wert 0 dieser Stellung wird für den Knoten B notiert.

Von der Wurzelstellung ausgehend werden in gleicher Weise die Zweige F... und J... untersucht. In den Knoten B, F und J stehen dann die Bewertungen der besten Endstellungen, die der Rechner bei jeweils optimalem Gegenzug von Schwarz erreichen kann. Der Rechner wählt als besten Zug den, der zur maximal bewerteten Stellung B führt.

Diese Methode, den besten Zug zu finden, nennt man Minimaxverfahren. Abwechselnd wählen die Spieler entsprechend ihrer Interessenlage die maximal (Weiß) oder die minimal (Schwarz) bewertete Folgestellung. Die Bewertung der Endstellungen (Gewinn > 0, Remis = 0, Verlust < 0) muß dabei aus der Sicht des Spielers vorgenommen werden, der in der Wurzelstellung am Zug ist.

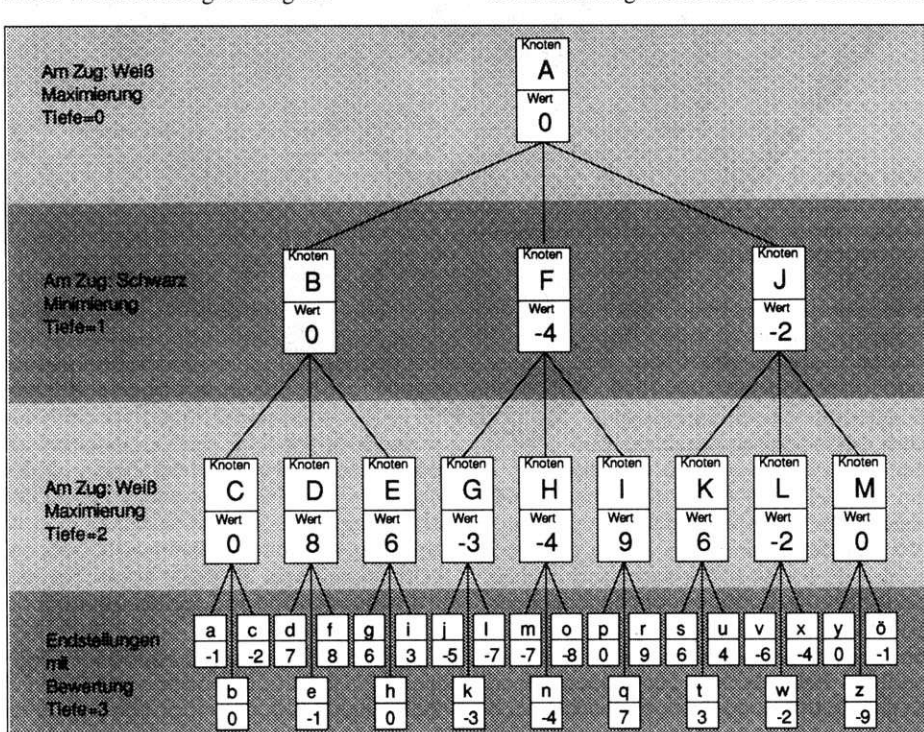


Bild 2: Bewertung der Spielbaumknoten mit dem Minimaxverfahren

Man kann das Minimaxverfahren natürlich nicht nur anwenden, um in der Anfangsstellung am Wurzelknoten des Spielbaumes den besten Zug zu finden. Wegen der Selbstähnlichkeit des Spielbaums kann man jeden Knoten einer Folgestellung wieder als Wurzel eines Unterbaumes ansehen. Das Minimaxverfahren findet also in jeder denkbaren Stellung eines Strategiespiels den besten Zug.

Die Komplexität von Spielbäumen realer Spiele wie Schach oder SOGO setzt der Anwendung des Minimaxverfahrens jedoch schnell Grenzen. Der SOGO-Spielbaum verzweigt sich in den Knoten in 16 Folgestellungen. Die Endknoten des Baums werden nach 64 Zügen erreicht, wenn alle Kugeln gesetzt sind. Man muß also mit 16 hoch 64 = 10 hoch 77 Endknoten rechnen. Diese Zahl wird nicht ganz erreicht, weil viele Partien schon nach weniger als 64 Zügen enden und weil nicht in allen Stellungen wegen bereits vollbesetzten Stiften 16 Züge möglich sind. Trotzdem bleibt eine gigantische Zahl möglicher Endstellungen.

Beim Schach rechnet man mit ca.10 hoch 120, bei Go gar mit 10 hoch 740 Endstellungen. Um auch nur über den ersten Zug zu entscheiden, müßte das Minimaxverfahren die Bewertung aller Endknoten verarbeiten. Das wäre selbst für die schnellsten bekannten Computer eine Arbeit von Jahrtausenden.

Heuristische Bewertung

Es gibt nur einen Ausweg: Man muß den Spielbaum vereinfachen. Einen Spielbaum mit 10 hoch 3 bis 10 hoch 4 Endknoten kann ein Computer wie der Amiga noch in angemessener Zeit mit dem Minimaxverfahren bearbeiten. Wenn in einem SOGO-Spielbaum die Endstellungen bereits in einer Tiefe von 3 Zügen lägen, wäre demnach die Auswertung von 16 hoch 3 = 4096 Endknoten noch zu bewältigen. Man erreicht das, indem man willkürlich alle Stellungen in der Tiefe 3 zu Endstellungen erklärt.

Das Minimaxverfahren erfordert jedoch für die Endknoten eine Bewertung nach Gewinn und Verlust. Bei echten Endstellungen ist es offensichtlich, welcher Spieler gewonnen hat. Bei den willkürlichen Endstellungen in der Tiefe 3 ist Gewinn oder Verlust noch nicht sicher vorherzusagen, das ist ja gerade der Reiz eines Strategiespiels. Man muß hier sinnvolle Kriterien für die Stellungsbewertung finden (griech.: heuriskein). Beim Schach könnte man die Anzahl der Figuren betrachten. Je mehr eigene Figuren je weniger gegnerischen gegenüberstehen, um so höher der Wert der Stellung. Beim SOGO ist eine Stellung günstig bewertet, in der viele potentielle Mühlen mit möglichst vielen eigenen Kugeln besetzt sind.

Die Spielstärke des Minimaxverfahrens auf dem verkürzten Spielbaum hängt stark ab von den Bewertungskriterien für die willkürlichen Endknoten. Der beste Zug wird nicht mehr mit 100%iger Sicherheit gefunden. Bei gleichen Bewertungskriterien – eine gewisse Güte dieser Kriterien sei vorausgesetzt – steigt die Spielstärke, wenn die willkürlichen Endstellungen in eine größere Tiefe verlegt werden. Der Rechner schaut dann im Spielgeschehen weiter voraus, muß allerdings eine höhere Zahl von Endknoten bearbeiten. Wie bereits bemerkt, setzt der AMIGA hier jedoch eine Grenze von ca. 4000.

Wieder bleibt nur ein Ausweg: der Spielbaum muß noch weiter vereinfacht werden.

Alfa/Beta-Pruning

Ohne weitere heuristische Zusatzannahmen, allein aufgrund logischer Überlegungen kann der Spielbaum noch weiter gestutzt werden (to prune: schneiden, stutzen). Wenn man den Suchvorgang des Minimaxverfahrens genauer untersucht, stellt man nämlich fest, daß die Bewertung einiger Knoten das Endergebnis gar nicht beeinflussen kann.

Man betrachte z.B. Stellung B in Bild 3. Das Minimaxverfahren übernimmt für diesen »min«-Knoten aus C, D und E den Wert der minimal bewerteten Stellung:

```
Wert(B) = min(Wert(C),Wert(D),Wert(E)).
```

Im ersten Schritt bestimmt der Rechner den Wert von Knoten C aus den Endknoten a, b und c. Wie in der Wurzel ist Weiß am Zug. Daher erhält der »max«-Knoten C den maximalen Wert der Folgestellungen a, b und c:

```
Wert(C) = max(Wert(a),Wert(b),Wert(c))
        = max(-1,0,-2) = 0
```

Bereits an dieser Stelle steht fest, daß B den Wert 0 nicht mehr überschreiten kann:

```
Wert(B) = min(Wert(C),Wert(D),Wert(E))
        = min( 0      ,Wert(D),Wert(E))<= 0.
```

Der Rechner speichert diese obere Grenze beta = 0 als vorläufigen Wert von B.

Wie kann die Bewertung von D und E diesen Wert von B noch beeinflussen? Wenn D und E mit einem Wert größer als beta bewertet werden, bleibt es bei dem Wert beta = 0. Nur wenn der Wert von D oder E kleiner als beta ausfällt, wird B noch geändert und erhält diesen kleineren Wert von D oder E.

Um die Frage zu entscheiden, ist die Stellung D zu bewerten. Hier ist wie in der Wurzel Weiß am Zug. Das Minimaxverfahren ordnet dem »max«-Knoten D daher den maximalen Wert der Endknoten d, e und f zu:

```
Wert(D) = max(Wert(d),Wert(e),Wert(f)).
```

Der erste Endknoten d liefert den Wert 7. Ganz egal, welche Werte die Endknoten e und f besitzen, der Wert von D kann nicht kleiner als 7 werden:

```
Wert(D) = max(7,Wert(e),Wert(f)) >= 7
```

Damit ist aber Wert(D) > beta und hat keinen Einfluß mehr auf die Bewertung von B:

```
Wert(B) = min(0, 7 o. größer, Wert(E)) <= 0
```

Die Auswertung der Endstellungen e und f kann an dieser Tatsache nichts ändern. Der Computer kann die Zeit für die Bewertung der Knoten sparen. Sobald an einem »max«-Knoten die Beta-Abbruch-Bedingung:

`Wert(»max«-Knoten) > beta`

erfüllt ist, kann also die weitere Auswertung des Knotens abgebrochen werden. Auch die Auswertung des »max«-Knotens E kann wegen der Beta-Bedingung

`Wert(E) >= 6 > beta`

schon nach dem Besuch des Endknotens g abgebrochen werden: B erhält den Wert 0.

Sobald an einem »min«-Knoten die alfa-Abbruch-Bedingung:

`Wert("min"-Knoten) < alfa`

erfüllt ist, kann die weitere Auswertung dieses Knotens abgebrochen werden. Über die Parameter alfa und beta informieren die Elternknoten, also ihre Folgeknoten, wann es sinnlos ist, einen Zweig weiter auszuwerten. Zu Beginn wird am Wuzelknoten die alfa/beta-Schere geöffnet, d.h. alfa = -unendlich und beta = +unendlich gesetzt. Alfa wird ausschließlich an »max«-Knoten verändert. Wenn die vom gerade bearbeiteten Folgeknoten zurückgelieferte Bewertung größer ist als der momentane alfa-Wert, wird alfa auf den Wert dieses Folgeknotens erhöht.

Bild 3: alfa/beta-Pruning beschneidet den Spielbaum nochmals

Eine entsprechende Überlegung wie für den »min«-Knoten B kann man auch für den »max«-Knoten A anstellen. B wird den Wert 0 liefern, soviel ist bereits bekannt. Da A ein »max«-Knoten ist, kann der Wert von A nicht mehr kleiner als dieser Wert 0 werden.

```
Wert(A)= max(Wert(B),Wert(F),Wert(J))
       = max( 0     ,Wert(F),Wert(J)) >= 0
```

Die untere Grenze alfa = 0 ist der vorläufige Wert von A. A wird nur dann die Werte von F oder J annehmen, wenn sie größer sind als alfa. Wenn die Werte von F und J kleiner sind als alfa, übernimmt A den Wert 0 von B.

Das Minimaxverfahren wertet als nächsten den »min«-Knoten F aus.

`Wert(F)=min(Wert(G),Wert(H),Wert(I))`

Die Endknoten j, k, l verleihen dem Knoten G den Wert -3. Da F ein »min«-Knoten ist, kann der Wert von F auch nach der Auswertung von H und I nicht mehr größer werden als der Wert -3 von G.

`Wert(F)=min(-3, Wert(H),Wert(J)) <= -3`

Damit ist Wert(F) < alfa und kann die Bewertung von Knoten A nicht beeinflussen:

`Wert(A)=max(0,-3 oder kleiner,Wert(J)) >=0.`

Die Auswertung der Zweige H und I ist also völlig überflüssig und kann ohne Informationsverlust unterbleiben.

Umgekehrt wird beta nur an »min«-Knoten modifiziert: Wenn die vom gerade bearbeiteten Folgeknoten gelieferte Bewertung kleiner ist als der momentane beta-Wert, wird beta auf den Wert dieses Folgeknotens erniedrigt. Alfa liefert den Grenzwert der Abbruch-Bedingung an »min«-Knoten, während beta diesen Grenzwert für »max«-Knoten darstellt. Die Tabelle gibt eine Übersicht des das spiegelbildlichen Verhaltens von alfa und beta.

Bedingung	Max-Knoten	Min-Knoten
W>α	α=W	———
W<α	———	α-Abbruch
W>β	β-Abbruch	———
W<β	———	β=W

Tabelle: Handhabung der alfa/beta-Schere; wann ist ein Knoten wichtig?

Alfa und beta werden von den Elternknoten an die folgenden Knoten weitergegeben. Eine Modifikation der Parameter an den Folgeknoten darf nicht den Wert von alfa oder beta bei den Elternknoten verändern, sonst wird möglicherweise gerade der Knoten mit der besten Bewertung abgeschnitten und der beste Zug wird nicht gefunden. Bei der Deklaration der weiter unten besprochenen Prozedur »Zug« tauchen daher alfa und beta als Wertparameter und nicht als variable Parameter in der Parameterliste auf.

Wenn wie beim Knoten F die Abbruchbedingung schon in geringer Tiefe greift, kann der Spielbaum ohne Informationsverlust um ganze Zweige gestutzt werden.

Ein Vergleich mit dem unbeschnittenen Spielbaum zeigt die Effektivität des alfa/beta-Prunings. Wenn sich jeder Knoten des Spielbaums in b Folgeknoten verzweigt und die Endknoten sich in der Tiefe t befinden, sind ohne alfa/beta-Pruning

$N0 = b^t$

Endknoten auszuwerten. Bei geradzahliger Baumtiefe kann das alfa/beta-Pruning diese Zahl reduzieren auf

$N(tgerade)=2*(b^{(t/2)})-1$

Bei ungeradzahliger Baumtiefe bleiben

$N(tungerade)=b^{((t+1)/2)} + b^{((t-1)/2)} -1$

auszuwertende Endknoten.

Die Spielbaumtiefe wird durch alfa/beta-Pruning also scheinbar halbiert. Bei gleicher Rechenzeit kann daher der Spielbaum bis zur doppelten Tiefe durchsucht werden; das bedeutet einen beträchtlichen Gewinn an Spielstärke. Die angegebenen Formeln für die Zahl der auszuwertenden Endknoten sind allerdings untere Grenzwerte, die in der Praxis nicht ganz erreicht werden. Das hängt damit zusammen, daß in ungünstig sortierten Spielbäumen die Möglichkeit eines Abbruchs erst erkannt wird, wenn die überflüssigen Zweige längst durchsucht worden sind.

Der Knoten J in Bild 3 zeigt dafür ein Beispiel. Die alfa-Abbruchbedingung greift erst nach der Bearbeitung des Folgeknotens L. Wenn L zuerst bearbeitet würde, entfielen sowohl Zweig K und M. Wird L aber erst nach der Auswertung von K oder M bearbeitet, wird zwar die Abbruchmöglichkeit erkannt, aber die Rechenzeit für die Zweige K oder M ist vergeudet. Das SOGO-Programm versucht, durch geschickte Sortierung des Spielbaums solche Situationen zu vermeiden.

... und die Praxis

Nach soviel grauer Theorie soll jetzt gezeigt werden, wie man Minimaxverfahren und alfa/beta-Pruning in einem konkreten Spielprogramm anwendet.

Programmstruktur:

Ein Strategie-Spielprogramm für den Amiga sollte vier Grundfunktionen besitzen:

- Intuition-Anwender-Oberfläche
- grafische Darstellung des Spielbretts und der Figuren
- Kontrolle des Spielablaufs
- Künstliche Intelligenz, um den besten Zug zu finden

Das in Modula-2 geschriebene Spielprogramm SOGO folgt diesem Schema und ist in vier Module gegliedert: ➪

❒ sogomain.mod enthält die Intuition-Oberfläche;
❒ sogograph.mod besorgt die Darstellung
❒ sogoctrl.mod kontrolliert den Spielablauf und die Einhaltung der Spielregeln;
❒ sogostrat.mod findet mit Minimaxverfahren und alfa/beta-Pruning den besten Zug;

Bild 4 zeigt den Zusammenhang der vier SOGO-Moduln. An der Spitze der Hierarchie steht das Modul sogomain. Hier werden beim Programmstart Screen und Windows geöffnet. Anschließend verwaltet sogomain die Intuition-Ereignisse. Wird beispielsweise ein Menüpunkt angeklickt, ruft sogomain eine Prozedur aus den Moduln sogoctrl oder sogograph auf, die das Gewünschte erledigt.

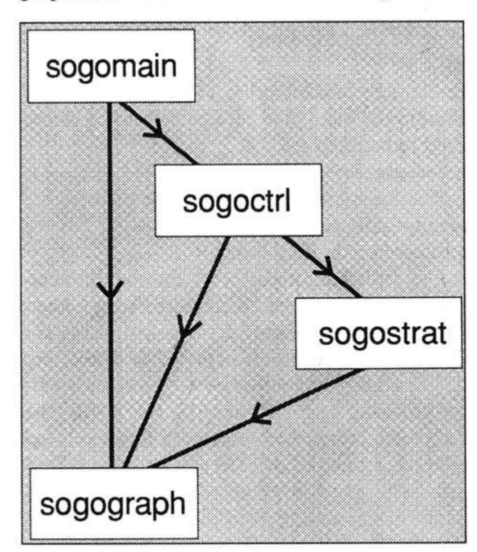


Bild 4: IMPORT von Prozeduren ist nur in Pfeilrichtung möglich

Die in Listing 1 bis 3 aufgelisteten Definitionsmoduln geben einen Überblick der implementierten Prozeduren. Die wichtigsten Menüfunktionen sind im Kasten »Was bietet das SOGO-Menü ?« beschrieben.

```
DEFINITION MODULE sogoctrl;
PROCEDURE BeginNewGame;
 (* startet eine neue SOGO-Partie *)
PROCEDURE Human;
 (* setzt ein Flag, sodaß der AMIGA nur das Spielbrett für
    2 menshl. Spieler liefert *)
PROCEDURE Computer;
 (* setzt ein Flag, sodaß SOGO gegen den AMIGA gespielt wird *)
PROCEDURE AutoPlay;
 (* setzt ein Flag, sodaß AMIGA gegen sich selbst spielt *)
PROCEDURE Black;
 (* Spieler 1 führt die weißen Kugeln, Spieler 2 die schwarzen *)
PROCEDURE White;
 (* Spieler 1 führt die schwarzen Kugeln, Spieler 2 die weißen *)
PROCEDURE LoadORSaveGame(LorS:INTEGER);
 (* lädt oder sichert eine SOGO-Partie *)
PROCEDURE TakeBackLastMove;
 (* nimmt einen Zug zurück *)
PROCEDURE DisplayStrings;
 (* schreibt die erweiterten Status-Infos ins Status-Window *)
PROCEDURE GameHandler;
 (* kontrolliert den Spielablauf *)
END sogoctrl.          © 1993 M&T
```

Listing 1: Das Modul »Sogoctrl.def« kontrolliert den Spielablauf

Den Spielablauf kontrolliert eine zyklisch von sogomain aus aufgerufene Prozedur gamehandler im Modul sogoctrl. Falls der Computer am Zug ist, aktiviert gamehandler den Strategiemodul sogostrat und setzt mit der Prozedur Zug die Künstliche Intelligenz des Amiga in Gang, die dann je Spielstärke einen mehr oder weniger guten Zug findet. Der gamehandler stellt diesen Zug mit der Prozedur DrawHiddenFigure aus dem Modul sogograph auf dem Bildschirm dar.

Wenn im nächsten Zyklus sogomain wiederum den gamehandler aufruft, bearbeitet dieser dann den Zug des Amiga-Gegenspielers. Über die Prozedur FigPicked aus sogograph erfährt der gamehandler, welcher SOGO-Stab angeklickt wurde, und stellt den gewünschten Zug, falls er den Spielregeln genügt, ebenfalls am Bildschirm dar.

Tips für Tüftler

Das Modul-Grundgerüst kann man im Prinzip für andere Strategiespiele praktisch 1:1 übernehmen. Da in allen Strategie-Spielprogrammen die gleichen Grundfunktionen realisiert werden müssen, werden sich die verschiedenen Programme zumindest im Hauptmodul (sogomain) und auf der Ebene der Definitionsmoduln sehr ähnlich sein. Die Unterschiede sind in den jeweiligen Implementationsmoduln realisiert. Das ist einleuchtend: eine Prozedur DrawBoard bringt in einem Schachspiel ein anderes Spielbrett auf den Bildschirm als im SOGO-Spiel.

Wenn man vom SOGO-Modul-Grundgerüst ausgehend ein anderes Strategiespiel programmieren will, sollte man mit der Intuition-Oberfläche beginnen. Die durch Menüklick aufzurufenden Funktionen kann man zunächst in den Implementationsmoduln als Dummyprozedur in der Form

```
PROCEDURE DrawBoard;
BEGIN
END DrawBoard;
```

schreiben. Man hat dann von Anfang an ein lauffähiges Programm, das in allen Phasen der Programmentwicklung getestet werden kann. Außerdem programmiert es sich leichter, wenn man immer wieder sichtbare Fortschritte erzielt.

```
DEFINITION MODULE sogograph;
FROM SYSTEM     IMPORT ADDRESS;
FROM Intuition  IMPORT ScreenPtr,WindowPtr,Menu;
FROM Graphics   IMPORT RastPortPtr;
TYPE Figtype        = (black,white,phantom,pin);
     ARRAY16        = ARRAY[1..16] OF INTEGER;
VAR  SpielScreenPtr : ScreenPtr;
     SpielWindowPtr : WindowPtr;
     SpielWinRPPtr  : RastPortPtr;
     SpielMenu      : ARRAY[0..2] OF Menu;
     StatusWindow   : WindowPtr;
     StatusWinRPPtr : RastPortPtr;
     Board          : ARRAY[1..64] OF Figtype;
     PotZug         : ARRAY16;
     Spieler        : Figtype;
     Rechner        : Figtype;
     AmZug          : Figtype;
     PutFig         : BOOLEAN;
     ToBeRemoved    : BOOLEAN;
PROCEDURE Okay(text:ARRAY OF CHAR;adr:ADDRESS):BOOLEAN;
 (* meldet Fehler, falls Windows oder Bitmaps nicht korrekt
    eingerichtet werden können *)
PROCEDURE SetColours;
 (* Farben-Handling *)
PROCEDURE Dim3D;
 (* setzt Flag, sodaß Spielbrett in Rot-Grün-3D erscheint *)
PROCEDURE Dim2D;
 (* setzt Flag, sodaß Spielbrett in perspektivischer 2-D-Ansicht
    erscheint *)
PROCEDURE DrawBoard;
 (* zeichnet das komplette Spielbrett in 2-D- oder 3-D-Ansicht *)
PROCEDURE DrawHiddenFigure(i:INTEGER;Typ:Figtype);
 (* zeichnet eine Kugel mit verdeckten Kanten ins Spielfeld *)
PROCEDURE DrawHiddenFigureRM(i:INTEGER;Typ:Figtype);
 (* nimmt eine Kugel vom Brett *)
PROCEDURE TypeFigure(Typ:Figtype);
 (* zeichnet eine Kugel der Farbe "Figtype" ins Statusfenster,
    um anzuzeigen, wer am Zug ist *)
PROCEDURE FigPicked(VAR BestZug:INTEGER):BOOLEAN;
 (* prüft, ob der menschl. Spieler Zug ausgeführt hat (TRUE)
    und gibt an, welcher Zug gewählt wurde (BestZug) *)
PROCEDURE RotateBoard;
 (* führt die gewünschte Drehung der Spielbrettansicht aus *)
PROCEDURE InitGame () :BOOLEAN;
 (* initialisiert die Grafik *)
PROCEDURE CloseDownGraph;
 (* gibt Speicher für die Bitmaps etc. zurück *)
END sogograph.              © 1993 M&T
```

Listing 2: »Sogograph.def« steuert die grafische Darstellung des Spiels

Was bietet das SOGO-Menü?

Menüpunkt	Funktion
- Menügruppe Playing:	
* Begin New Game	nimmt Kugeln vom Brett und beginnt eine Partie. Weiß hat den ersten Zug.
* Choose Your Opponent	wählt Spielpartner. Human bedeutet ein Spiel mit einem menschlichen Gegenüber. Computer wählt den Amiga als Gegenspieler. AutoPlay läßt den Amiga gegen sich selbst spielen.
* Choose Level	legt die Spielstärke des Amiga fest. Seine Antwortzeit nimmt mit wachsendem Level von weniger als 1 s (Level 1) bis auf einige Minuten (Level 5) zu. Level 6 und 7 können auf einem normalen AMIGA mehrere Stunden Bedenkzeit benötigen.
* Choose Side	wählt die Kugelfarbe des 1. Spielers.
* Save This Game	sichern eines Spiels
* Load A Saved Game	lädt ein gesichertes Spiel und setzt die Kugeln auf das Spielbrett.
* Quit	beendet das SOGO-Programm.
Menügruppe View:	
* Set Colours	öffnet ein Fenster mit Gadgets zur Änderung der Farbeinstellungen.
* Type of View	wählt zwischen 2-D- und 3-D-Darstellung (Rot-Grün-Filterbrille erforderlich)
* Rotate Board	öffnet ein Fenster für die Einstellung des Blickwinkels auf das Spielbrett.
- Menügruppe Options	
* Take Back Last Move	nimmt die zuletzt gesetzte Kugel wieder vom Brett.
* Status-Info	wählt aus, ob im Status-Window nicht nur der aktuelle Spielzustand sondern auch der Gedankengang des AMIGA bei der Suche nach dem besten Zug dargestellt werden soll.

Das Status-Window enthält folgende Informationen:
+ wer ist am Zug;
+ den Spielausgang (Gewinner oder "REMIS");
+ ein horizontales Balkendiagramm; die Zahl der Balken richtet sich nach der gewählten Suchtiefe im Spielbaum; jedem Balken entspricht eine bestimmte Tiefenstufe
+ die vorgegebene Spielstärke, der die Zahl der vorausberechneten Halbzüge entspricht.
+ Zahl der durch die alpha/beta-Bedingung abgeschnittenen Zweige des Spielbaumes;
+ Gesamtzahl der bewerteten Spielstellungen;

```
DEFINITION MODULE sogostrat;
FROM sogograph IMPORT Figtype,ARRAY16;
FROM Intuition IMPORT IDCMPFlagSet;

TYPE  MUEHLEN   =  ARRAY[1..76],[1..6] OF INTEGER;
      ZUEGE     =  ARRAY[1..64],[1..20] OF INTEGER;
      Stat      =  ARRAY[1..8] OF INTEGER;
CONST DiagBonus = 5;
VAR
MHL    : MUEHLEN;
 (* MHL[X][1..4]: Kugelpositionen der Mühle Nr. X              *)
 (* MHL[X][5]   : Anzahl der weißen Kugeln in Mühle X          *)
 (* MHL[X][6]   : Anzahl der schwarzen Kugeln in Mühle X       *)
ZK    : ZUEGE;
 (* ZK[Y][1]    : Zahl der Mühlen, in denen Zug Y vorkommt.   *)
 (* ZK[Y][2..8] : Die Nummern X der Mühlen,in denen Y vorkommt *)
 (* ZK[Y][9]    : frei            *)
 (* ZK[Y][10]   : Anzahl der Mühlen von Y mit 1 weißen Kugel   *)
 (* ZK[Y][11]   : Anzahl der Mühlen von Y mit 2 weißen Kugeln  *)
 (* ZK[Y][12]   : Anzahl der Mühlen von Y mit 3 weißen Kugeln  *)
 (* ZK[Y][13]   : Anzahl der Mühlen von Y mit 4 weißen Kugeln  *)
 (* ZK[Y][14]   : frei                                         *)
 (* ZK[Y][15]   : Anzahl der Mühlen von Y mit 1 schwarzen Kugel*)
 (* ZK[Y][16]   : Anzahl der Mühlen von Y mit 2 schw. Kugeln *)
 (* ZK[Y][17]   : Anzahl der Mühlen von Y mit 3 schw. Kugeln *)
 (* ZK[Y][18]   : Anzahl der Mühlen von Y mit 4 schw. Kugeln *)
 (* ZK[Y][19]   : frei                                         *)
 (* ZK[Y][20]   : frei                                         *)
MHLStat    : Stat;
 (* MHLStat enthält die Zahl der von jedem Spieler ausschließlich
     mit Kugeln seiner Farbe belegten Mühlen;
     MHLStat[i]   : Zahl der Mühlen mit i weißen Kugeln, i=1..4
     MHLStat[4+j] : Zahl der Mühlen mit j schw. Kugeln, j=1..4 *)
depth                            : INTEGER;
 (* Im Menu festgelegte Suchtiefe                  *)
vardepth                         : INTEGER;
 (* tatsächliche Suchtiefe; in den ersten Zügen
    ist vardepth = 1,  egal wie depth gesetzt ist  *)
BestZug                          : INTEGER;
 (* Zug mit der besten Bewertung                   *)
gesamtAnz,betaAnz,alphaAnz,rndm : LONGINT;
 (* Status-Infos *)
Verl                             : Figtype;
 (* erhält Farbe d. Verlierers *)
ZugListe                         : ARRAY[1..64] OF INTEGER;
 (* enthält die Zughistorie, wird benötigt beim Sichern eines
    Spiels und bei der Zugrücknahme                *)
ZLIndex                          : INTEGER;
 (* aktueller Index v. ZugListe *)
interrupt,StatusInfo             : BOOLEAN;
class                            : IDCMPFlagSet;
code                             : CARDINAL;
PROCEDURE SetStatusInfo(i:INTEGER);
 (* normale (i=0) oder erweiterte (i=1) Statusanzeige  *)
PROCEDURE SetDepth(i:INTEGER);
 (* definiert die im Menü gewählte Suchtiefe *)
PROCEDURE Zug(VAR wert:LONGINT; farbe:Figtype;
            step:INTEGER; alpha,beta:LONGINT;
            PotZugUnSorted:ARRAY16);
 (* sucht im Spielebaum den besten Zug *)
PROCEDURE InitStrat;
 (* initialisiert die Variablen des Moduls sogostrat *)
END sogostrat.    © 1993 M&T
```

»Listing 3: Sogostrat« – der »Kern der Künstlichen Intelligenz« des Spiels

Läuft die Intuition-Menü-Oberfläche fehlerfrei, kann man als nächstes die Dummyprozedur von DrawBoard mit sinnvollem Programmcode füllen, sodaß der Aufruf dieser Prozedur auch tatsächlich das gewünschte Spielbrett auf den Bildschirm zeichnet.

Nach und nach arbeitet man sich so zum Kern zu den Strategieroutinen vor. Die härteste Nuß, die Programmierung der Prozedur für die Bewertung der willkürlichen Endstellungen, spart man sich bis ganz zuletzt auf. Diese Prozedur bestimmt wesentlich die Spielstärke und erfordert eine gründliche Spielanalyse. Wer ein Strategiespiel programmieren will, sollte dieses Spiel sehr gut beherrschen. Man wird mit einfachen Bewertungskriterien beginnen und findet dann in vielen Testspielen gegen den Computer weitere Ansätze zur Optimierung der heuristischen Stellungsbewertung. Die bereits vorhandene Anwenderoberfläche mit einer übersichtlichen Spielbrettdarstellung sowie den bereits implementierten Funktionen wie »Zug setzen«, »Zug zurücknehmen«, »Seitenwechsel", »Spiel sichern«, »Spiel laden« sind eine unschätzbare Hilfe, weil man sich bei der Suche nach besseren Bewertungskriterien ganz auf das Spiel konzentrieren kann.

Das SOGO-Programm mit allen Moduln umfaßt einige Tausend Zeilen Quellcode, so daß ein kompletter Abdruck nicht sinnvoll ist. Um Ihnen das Abtippen zu ersparen, veröffentlichen wir das lauffähige SOGO-Programm mit allen Quellcodes auf der PD-Diskette zu diesem Heft. Nur das Strategiemodul sogostrat.mod ist in Listing 4 abgedruckt. Es enthält Minimaxverfahren und alfa/beta-Pruning als Modula-2-Programmcode.

Datenstrukturen:

Beim Programmstart werden alle Variablen des Moduls sogostrat in der Prozedur InitStrat mit Anfangswerten belegt. Das Spielbrett ist durch das ARRAY Board dargestellt. Es enthält entsprechend den 64 möglichen Kugelpositionen 64 Elemente, die den Wert pin, black oder white annehmen können. In Bild 10 und 11 trägt jede Kugelposition die Nummer des zugehörigen ARRAY-Elementes.

Die Mühlenstruktur des Spielbretts wird bei der Initialisierung der ARRAYs MHL und ZK festgelegt. MHL enthält für jede der insgesamt 76 möglichen Mühlen die zugehörigen 4 Kugelpositionen. Umgekehrt gibt ZK für jede der 64 Kugelpositionen an, in wieviel und in welchen der 76 möglichen Mühlen sie nthalten ist. Während des Spiels wird in MHL und ZK sowie MHLStat nach jedem Zug die Datenbasis für die Stellungsbewertung auf den neuesten Stand gebracht.

In MHL wird für jede der 76 möglichen Mühlen notiert, mit wieviel Kugeln jeder Spieler sie belegt hat. In ZK wird für jede Kugelposition eingetragen, wieviele der zugehörigen möglichen Mühlen jeder Spieler mit wieviel eigenen Kugeln beherrscht. Eine mögliche Mühle gilt dann als von einem Spieler beherrscht, wenn sie ausschließlich mit Kugeln seiner Farbe belegt ist.

MHLStat enthält für beide Spieler die Gesamtanzahl der jeweils mit 1, 2, 3 oder 4 Steinen beherrschten Mühlen.

Minimax und alfa-beta

Wenn der Amiga an der Reihe ist, sucht er mit der Prozedur Zug im SOGO-Spielbaum nach dem besten Zug. Die in der aktuellen Stellung möglichen Züge werden im Feld PotZugUnSorted an diese Prozedur übergeben. Sie setzt eine Kugel probeweise aufs Spielbrett und ruft sich anschließend selbst erneut auf. Bis zu welcher Tiefe der Spielbaum durch diese Rekursion entwickelt wird, bestimmt die Variable step, die bei jedem Rekursionsschritt um 1 verringert wird.

Ist step = 0, wird anstelle eines weiteren Rekursionsschritts die erzeugte Stellung bewertet. In jeder Rekursionsstufe werden die von den folgenden Stufen zurückgelieferten Werte mit dem Minimaxverfahren bearbeitet. Die alfa/beta-Bedingungen entscheiden, ob Probezüge gespart werden können.

Auf der ersten Rekursionsstufe wird schließlich anhand des günstigsten, zurückgelieferten Wertes der beste Zug für die aktuelle Stellung bestimmt.

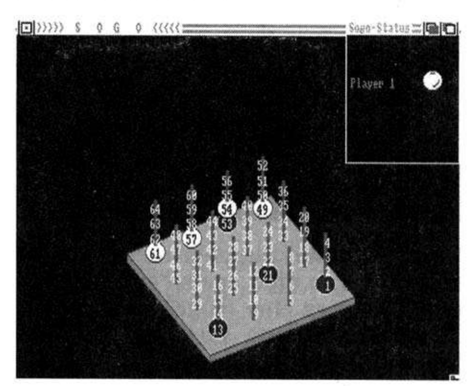


Bild 10: Weiß ist am Zug. Es besteht Kreuzmühlengefahr für Weiß. Wenn Weiß nicht eine der schwarzen Mühlen (1,5,9,13) oder (5,21,37,53) blockiert, kann Schwarz im Gegenzug Position 5 besetzen. Der Gewinn von Schwarz ist dann nicht aufzuhalten.

Um das alfa/beta-Pruning effektiv zu machen, werden die möglichen Züge vor jedem weiteren Rekursionsschritt mit der Prozedur SortPotZug sortiert. Wenn ein Spieler eine Mühle bereits mit mehreren Kugeln beherrscht, wird er versuchen, diese Mühle weiter auszubauen, um schließlich mit der vierten Kugel seine Mühle zu vollenden und zu gewinnen. Andererseits wird sein Gegenspieler alles tun, um das zu verhindern. Er wird ebenfalls eine Kugel in der vom Gegner beherrschten Mühle plazieren wollen, um dessen Gewinn zu verhindern. Es ist also wahrscheinlich, daß ein Zug, der zu einer mit bereits vielen Kugeln beherrschten Mühle gehört, der beste Zug in einer Stellung ist.

Wenn man die in einer Stellung möglichen Züge entsprechend sortiert, kann man den Zug mit der voraussichtlich besten Bewertung zuerst ausprobieren. Oft werden dann bereits im ersten Probezug die alfa- oder beta-Bedingung erfüllt und die restlichen Züge brauchen nicht überprüft zu werden.

Bewertungskriterien

Der Amiga kann den SOGO-Spielbaum in erträglicher Zeit auch bei alfa/beta-Pruning nur etwa 7 Züge tief durchsuchen. Für die in dieser Tiefe angetroffenen willkürlichen Endstellungen müssen Bewertungskriterien formuliert werden. Ziel ist, jeder Stellung eine Zahl zuzuordnen, die ihren Wert wiedergibt.

Die Prozedur BEWERTUNG untersucht zuerst das Spielbrett auf bestimmte Stellungsmuster. Dabei wird z.B. festgestellt, ob Kreuzmühlengefahr (Bild 10) besteht, eine

Doppeldrohung (Bild 11) aufgebaut wird oder im nächsten Zug eine Mühle vollendet werden kann. Für insgesamt 20 Kriterien dieser Art wird geprüft, wie oft sie in der zu bewertenden Stellung erfüllt sind.

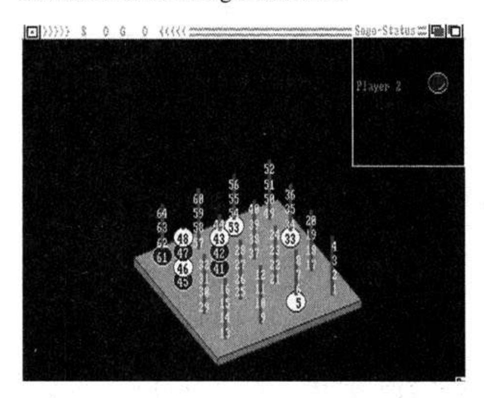


Bild 11: Eine Doppeldrohung gegen Schwarz. Um die Lage zu entschärfen, muß Schwarz eine Kugel auf Position 21 setzen. Könnte Weiß die Position besetzen, würden die Mühlen (5,21, 37,53) und (33,38,43,48) mit je drei Kugeln von Weiß beherrscht. Die fehlenden vierten Kugeln 37 und 38 liegen übereinander. Wenn Schwarz mit einer Kugel an Position 37 die untere Mühle verhindert, kann Weiß die obere vervollständigen.

Die den verschiedenen Kriterien zugehörigen Häufigkeiten werden in den ARRAYS ZugStat bzw. MHLStat gespeichert. Der endgültige Wert einer Stellung ist die gewichtete Summe der Häufigkeiten. Durch die Gewichtung wird der unterschiedlichen Bedeutung der Kriterien Rechnung getragen. Die Faktoren sind im ARRAY GEWICHT gespeichert.

Die Prozedur BEWERTUNG liefert für eindeutige Gewinnpositionen sehr große positive Werte, ausgeglichene Stellungen erhalten Werte um 0 und Verlustpositionen werden extrem negative Werte zugewiesen.

Spielprogramme mit heuristischen Bewertungskriterien werden eigentlich nie fertig. Obwohl SOGO in der vorliegenden Version eine passable Spielstärke besitzt, gibt es Verbesserungsansätze: Man könnte mit einer Evolutionsstrategie die Künstliche Intelligenz sogar dazu bringen, sich selbst zu optimieren, indem man den Amiga z.B. die Auswirkung zufälliger Mutationen am ARRAY GEWICHT untersuchen läßt.

Dazu läßt man den Amiga 100 Partien gegen sich selbst spielen. Als Spieler 1 verwendet er das mutierte ARRAY, als Spieler 2 das ursprüngliche. Das ARRAY des Gewinners überlebt und ist Ausgangsbasis für neue Mutationen. Wenn man dem AMIGA nur genügend Zeit läßt, wird er irgendwann einmal als Spieler 1 gewinnen und damit eine Verbesserung für das ARRAY GEWICHT gefunden haben. Ein Tip für den, der es probieren will: Elemente für eine Erfolg versprechende Mutation sind: GEWICHT[8..10] und GEWICHT[18..20]. *ub*

```
IMPLEMENTATION MODULE sogostrat;
FROM SYSTEM       IMPORT ADR,LONGSET;
FROM Intuition    IMPORT IntuiMessagePtr,IntuiText,IDCMPFlagSet;
FROM Exec         IMPORT SetSignal,GetMsg,ReplyMsg;
FROM InOut        IMPORT WriteInt,WriteLn,WriteString;
FROM Conversions  IMPORT ValToStr;
FROM Str          IMPORT Length;
FROM RandomNumber IMPORT RND;
FROM Graphics     IMPORT SetAPen,Draw,Move,Text,jam1;
FROM sogograph    IMPORT Figtype,Board,AmZug,SpielWinRPPtr,
                         SpielWindowPtr,
    StatusWinRPPtr,StatusWindow,ARRAY16;
VAR S10   : ARRAY[1..10] OF CHAR;
 err   : BOOLEAN;
     IntuiMsg  : IntuiMessagePtr;
 gewicht   : ARRAY[1..20] OF LONGINT;
PROCEDURE SortPotZug(VAR
PotZugSorted:ARRAY16;PotZugUnSorted:ARRAY16);
(* sortiert die möglichen Züge, sodaß das alfa/beta-      *)
(* Pruning effektiver wird;                               *)
(* Sortierkriterium: Je weiter die zu einem Zug gehörenden
Mühlen*)
(* ausgebaut sind, um so eher wird dieser Zug untersucht.  *)
VAR i,j,k,m :INTEGER;
    PotZugIndex :ARRAY[1..16] OF INTEGER; (* enthält Bewertung
der
   potentiellenZüge entsprechend Sortierkriterium *)
BEGIN
  FOR i:=1 TO 16 DO
    PotZugIndex[i]:=0;
    PotZugSorted[i]:=PotZugUnSorted[i];
  END;
  (* Bewertung der Züge nach Sortierkriterium *)
  FOR i:=1 TO 16 DO
   IF PotZugUnSorted[i]>0 THEN
    FOR j:=2 TO ZK[PotZugUnSorted[i]][1]+1 DO
      k:=ZK[PotZugUnSorted[i]][j];
      IF (MHL[k][5]=0) OR (MHL[k][6]=0) THEN
         IF (MHL[k][5]=3) OR (MHL[k][6]=3) THEN
            PotZugIndex[i]:=PotZugIndex[i]+5000;
         ELSE
            IF (MHL[k][5]=2) OR (MHL[k][6]=2) THEN
               PotZugIndex[i]:=PotZugIndex[i]+500;
     ELSE
        IF (MHL[k][5]=1) OR (MHL[k][6]=1) THEN
           PotZugIndex[i]:=PotZugIndex[i]+20;
        ELSE
           IF (MHL[k][5]=0) AND (MHL[k][6]=0) THEN
              PotZugIndex[i]:=PotZugIndex[i]+1;
    END;
               END;
            END;
         END;
      END;
    END;(* of FOR *)
   END; (* of IF *)
  END;(* of FOR *)
  (* SHELL-Sort von PotZugUnsorted in PotZugSorted *)
  k:= 8;
  WHILE (k>0) DO
    i:=k;
    WHILE (i<16) DO
      j:=i-k;
      WHILE (j>=0) AND (PotZugIndex[j+1] < PotZugIndex[j+k+1]) DO
        m:= PotZugIndex[j+1];
        PotZugIndex[j+1]:=PotZugIndex[j+k+1];
        PotZugIndex[j+k+1]:=m;
        m:=PotZugSorted[j+1];
        PotZugSorted[j+1]:=PotZugSorted[j+k+1];
        PotZugSorted[j+k+1]:=m;
        j:=j-k;
      END;
      i:=i+1;
    END;
    k:=k DIV 2;
  END;
END SortPotZug;
PROCEDURE SetStatusInfo(i:INTEGER);
(* setzt (i=1) oder löscht (i=0) das StatusInfo-Flag, das die An-
  zeige von Spielstatus-Informationen im Status-Window steuert *)
BEGIN
  IF i=0 THEN StatusInfo:=FALSE ELSE StatusInfo:=TRUE END;
END SetStatusInfo;
PROCEDURE SetDepth(i:INTEGER);
(* Legt allgemeine Suchtiefe und Bewertungsgewichte fest *)
BEGIN
  gewicht[1]:= 10000000;
  gewicht[2]:=   900000;
  gewicht[3]:=    80000;
  gewicht[4]:=   800000;
  gewicht[5]:=    20000;
  gewicht[6]:=   500000;
  gewicht[7]:=    90000;
  gewicht[8]:=        3;
  gewicht[9]:=        4;
  gewicht[10]:=       5;
  gewicht[11]:=       0;
  gewicht[12]:=- 700000;
  gewicht[13]:=-   1000;
  gewicht[14]:=-  18000;
  gewicht[15]:=-   5000;
  gewicht[16]:=-  16000;
  gewicht[17]:=-    500;
  gewicht[18]:=-      3;
  gewicht[19]:=-      4;
  gewicht[20]:=-      5;
  depth:=i;
  IF (i=0) THEN
        gewicht[1]:= 10000000;
   gewicht[2]:=        0;
   gewicht[3]:=        0;
   gewicht[4]:=   800000;
   gewicht[5]:=   0;
   gewicht[6]:=   0;
   gewicht[7]:=   0;
   gewicht[8]:=        3;
   gewicht[9]:=        4;
   gewicht[10]:=       5;
   gewicht[11]:=-      0;
   gewicht[12]:=-      0;
   gewicht[13]:=-      0;
   gewicht[14]:=-  18000;
   gewicht[15]:=-  0;
   gewicht[16]:=-  0;
   gewicht[17]:=-  0;
   gewicht[18]:=-      3;
   gewicht[19]:=-      4;
   gewicht[20]:=-      5;
  END;
END SetDepth;
PROCEDURE Bewertung(farbe :Figtype;PotZug:ARRAY16):LONGINT;
(* führt die Bewertung einer willkürlichen Endstellung durch.*)
(* farbe : gibt am Zug befindlichen Spieler an, aus dessen Sicht *)
(*         die Bewertung durchgeführt werden soll.          *)
(* PotZug: die in der zu bewertenden Stellung möglichen Züge *)
(* RETURN: wert: LONGINT-Wert gibt den Stellungswert an        *)
VAR H1,H2,i,j,k,m,n,p,q  :INTEGER; (* Hilfs- und Laufvariablen *)
    wert   :LONGINT; (* Stellungsbewertung           *)
    ZugNr,ZugNrUp,ZugNrUpUp :INTEGER; (* Nummern von 3
übereinander-
                              liegenden Kugelpositionen *)
    ZKME,ZKYOU,ME,YOU  :INTEGER; (* Indexoffsets für ZK und MHL *)
    MEMSInd,YOUMSInd  :INTEGER; (* Indexoffset für MHLStat     *)
    ZugStat   :ARRAY[1..17] OF INTEGER;
   (* Jedes Feldelement von ZugStat entspricht einem Bewertungskriterium.
    Ist es erfüllt, wird das Element inkrementiert. Sind alle
    Bewertungskriterien überprüft, ergibt die gewichtete Addition
    der Elemente von ZugStat die Stellungsbewertung.         *)
  PotDDUp,PotDDDown  :BOOLEAN; (* Flags, die beim Erkennen einer *)
  PotPotDDUp,PotPotDDDown :BOOLEAN; (*Doppeldrohung gesetzt werden*)
    Damokles   :BOOLEAN; (* Flag wird gesetzt, wenn im
                          Gegenzug Spielverlust droht    *)
BEGIN
(* Infos für das Status-Window *)
  INC(gesamtAnz);
  IF StatusInfo THEN
     IF gesamtAnz MOD 8 = 0 THEN
       ValToStr(gesamtAnz,FALSE,S10,10,8," ",err);
       Move(StatusWinRPPtr,80,75);
       Text(StatusWinRPPtr,ADR(S10),Length(S10));
     END;
  END;
(* Initialisieren *)
  wert:=0;
  FOR i:=1 TO 17 DO ZugStat[i]:=0; END;
(* Indexoffsets festlegen *)
  IF (farbe = white) THEN
```

```
    ME:=5; YOU:=6; ZKME:=10; ZKYOU:=15; MEMSInd:=1; YOUMSInd:=5;
  ELSE
    ME:=6; YOU:=5; ZKME:=15; ZKYOU:=10; MEMSInd:=5; YOUMSInd:=1;
  END;
(* Hauptschleife: Auswirkungen der Folgezüge prüfen *)
  FOR i:=1 TO 16 DO
(* Feststellen, ob Kreuzmühlengefahr oder Doppeldrohung vorliegt
*)
    ZugNr:=PotZug[i];
(*    WriteInt(ZugNr,4);*)
    IF ZugNr>0 THEN
      IF ZugNr MOD 4 > 0 THEN
        ZugNrUp:=ZugNr+1;
      ELSE
        ZugNrUp:=0;
      END;
      IF ZugNrUp MOD 4 > 0 THEN
        ZugNrUpUp:=ZugNrUp+1;
      ELSE
        ZugNrUpUp:=0;
      END;
      PotDDUp:=FALSE;PotDDDown:=FALSE;
      PotPotDDUp:=FALSE;PotPotDDDown:=FALSE;
      IF (ZK[ZugNr][ZKME]>0) OR (ZK[ZugNr][ZKME+1]>0) THEN
        H1:=ZK[ZugNr][1]+1;
        m:=1;
        FOR j:=2 TO H1 DO
            IF MHL[ZK[ZugNr][j]][ME]=2 THEN
              IF MHL[ZK[ZugNr][j]][YOU]=0 THEN
                FOR k:=1 TO 4 DO
                  H2 := MHL[ZK[ZugNr][j]][k];
                  IF (Board[H2]=pin) AND (H2#ZugNr) THEN
                    FOR n:=1 TO 16 DO
                      IF PotZug[n]=H2 THEN
                        INC(m);
                      END;
                    END;
                    IF ZK[ZugNr][j]>16 THEN
                      IF H2 MOD 4 >0 THEN
                        IF ZK[H2+1][ZKME+2]>0 THEN
      PotDDUp:=TRUE;
          END;
        END;
        IF ((H2-1) MOD 4) >0 THEN
                        IF ZK[H2-1][ZKME+2]>0 THEN
      PotDDDown:=TRUE;
          END;
        END;
    END;
        END;
    END;
  END;
        END;
      IF ZK[ZugNr][j]>16 THEN
              IF MHL[ZK[ZugNr][j]][ME]=1 THEN
                IF MHL[ZK[ZugNr][j]][YOU]=0 THEN
                  FOR k:=1 TO 4 DO
                    H2 := MHL[ZK[ZugNr][j]][k];
                    IF (Board[H2]=pin) AND (H2#ZugNr) THEN
                      IF H2 MOD 4 >0 THEN
                        IF ZK[H2+1][ZKME+2]>0 THEN
      PotPotDDUp:=TRUE;
          END;
        END;
        IF ((H2-1) MOD 4) >0 THEN
                        IF ZK[H2-1][ZKME+2]>0 THEN
      PotPotDDDown:=TRUE;
          END;
        END;
            END;
        END;
    END;
        END;
      END;
  END;
        END;
        DEC(m);
(* Ist die in der Position oberhalb von ZugNr liegende Kugel die
4.
 in einer Mühle des Gegenspielers, sodaß er im Gegenzug gewinnt?
*)
        Damokles:=FALSE;
        IF ZugNrUp>0 THEN
          IF ZK[ZugNrUp][ZKYOU+2]>0 THEN
            Damokles:=TRUE;
          END;
        END;
(* 4. Stein in ME-MHL setzbar: ME gewinnt *)
        IF ZK[ZugNr][ZKME+2]>0 THEN
          INC(ZugStat[1]);
        END;
(* Doppeldrohung zu Spielgewinn wandelbar: ME gewinnt *)
        IF ZugNrUp>0 THEN
          IF ZugNrUpUp>0 THEN
            IF (ZK[ZugNrUp][ZKME+2]>0) THEN
              IF (ZK[ZugNrUpUp][ZKME+2]>0) THEN
                INC(ZugStat[2]);
              END;
            END;
          END;
        END;
        IF NOT Damokles THEN
(* Kreuzungsstein in ME-MHL mit je 2 Steinen setzbar *)
          IF (ZK[ZugNr][ZKME+1]>1)  THEN
            INC(ZugStat[3]);
(* Kreuzungsstein in ME-MHL mit je 2 Steinen setzbar; in beiden
MHL
 kann im nächsten Zug der 4. Stein gesetzt werden: ME gewinnt *)
            IF (m>1) THEN
              INC(ZugStat[4]);
            END;
          END;
(* mit 2. Stein in ME-MHL Doppeldrohung gegen YOU vorbereiten *)
          IF PotPotDDUp OR PotPotDDDown THEN
            INC(ZugStat[5]);
          END;
(* mit 3.Stein in ME-MHL Doppeldrohung gegen YOU möglich *)
          IF PotDDUp OR PotDDDown THEN
            INC(ZugStat[6]);
          END;
(* Stein in der Kreuzung von mehr als 1 1er-MEMHL und 1 2er MEMHL
*)
          IF (ZK[ZugNr][ZKME]>1) AND (ZK[ZugNr][ZKME+1]>0) THEN
            INC(ZugStat[7]);
          END;
        END;
(* Dasselbe für YOU-MHL *)
        PotDDUp:=FALSE;PotDDDown:=FALSE;
        PotPotDDUp:=FALSE;PotPotDDDown:=FALSE;
        IF (ZK[ZugNr][ZKYOU]>0) OR (ZK[ZugNr][ZKYOU+1]>0) THEN
          H1:=ZK[ZugNr][1]+1;
          m:=1;
          FOR j:=2 TO H1 DO
              IF MHL[ZK[ZugNr][j]][YOU]=2 THEN
                IF MHL[ZK[ZugNr][j]][ME]=0 THEN
                  FOR k:=1 TO 4 DO
                    H2 := MHL[ZK[ZugNr][j]][k];
                    IF (Board[H2]=pin) AND (H2#ZugNr) THEN
                      FOR n:=1 TO 16 DO
                        IF PotZug[n]=H2 THEN
                          INC(m);
                        END;
                      END;
        IF ZK[ZugNr][j]>16 THEN
                      IF H2 MOD 4 >0 THEN
                        IF ZK[H2+1][ZKYOU+2]>0 THEN
        PotDDUp:=TRUE;
          END;
        END;
        IF ((H2-1) MOD 4) >0 THEN
                        IF ZK[H2-1][ZKYOU+2]>0 THEN
        PotDDDown:=TRUE;
          END;
        END;
      END;
          END;
        END;
      END;
          END;
        IF ZK[ZugNr][j]>16 THEN
                IF MHL[ZK[ZugNr][j]][YOU]=1 THEN
                  IF MHL[ZK[ZugNr][j]][ME]=0 THEN
                    FOR k:=1 TO 4 DO
                      H2 := MHL[ZK[ZugNr][j]][k];
                      IF (Board[H2]=pin) AND (H2#ZugNr) THEN
                        IF H2 MOD 4 >0 THEN
                          IF ZK[H2+1][ZKYOU+2]>0 THEN
        PotPotDDUp:=TRUE;
          END;
            END;
            IF ((H2-1) MOD 4) >0 THEN
                          IF ZK[H2-1][ZKYOU+2]>0 THEN
          PotPotDDDown:=TRUE;
              END;
            END;
              END;
            END;
          END;
        END;
            END;
          END;
  END;
          END;
          DEC(m);
          Damokles:=FALSE;
          IF ZugNrUp>0 THEN
            IF ZK[ZugNrUp][ZKME+2]>0 THEN
              Damokles:=TRUE;
            END;
          END;
(* 4. Stein in YOU-MHL setzbar: YOU erzeugt Zugzwang für ME *)
        IF ZK[ZugNr][ZKYOU+2]>0 THEN
          INC(ZugStat[11]);
        END;
(* Doppeldrohung zu Spielgewinn wandelbar: YOU gewinnt *)
        IF ZugNrUp>0 THEN
          IF (ZK[ZugNrUp][ZKYOU+2]>0) THEN
            IF (ZK[ZugNr][ZKYOU+2]>0) THEN
              INC(ZugStat[12]);
            END;
          END;
        END;
        IF NOT Damokles THEN
(* Kreuzungsstein in YOU-MHL mit je 2 Steinen setzbar *)
          IF (ZK[ZugNr][ZKYOU+1]>1)  THEN
            INC(ZugStat[13]);
(* Kreuzungsstein in YOU-MHL mit je 2 Steinen setzbar; in beiden MHL
 kann im nächsten Zug der 4. Stein gesetzt werden: Zugzwang *)
            IF (m>1) THEN
              INC(ZugStat[14]);
            END;
          END;
(* mit 2. Stein in YOU-MHL Doppeldrohung gegen ME vorbereiten *)
      (* IF PotPotDDUp OR PotPotDDDown THEN *)
          IF PotDDUp OR PotDDDown THEN
            INC(ZugStat[15]);
          END;
(* mit 3.Stein in YOU-MHL Doppeldrohung gegen YOU möglich *)
      (* IF PotDDUp OR PotDDDown THEN *)
          IF ZugNrUp>0 THEN
            IF ZugNrUpUp>0 THEN
              IF (ZK[ZugNrUp][ZKYOU+2]>0) THEN
                IF (ZK[ZugNrUpUp][ZKYOU+2]>0) THEN
                  INC(ZugStat[16]);
                END;
              END;
            END;
          END;
(* Stein in der Kreuzung von mehr als 1 1er-YOUMHL und 1 2er YOUMHL*)
          IF (ZK[ZugNr][ZKYOU]>1) AND (ZK[ZugNr][ZKYOU+1]>0) THEN
            INC(ZugStat[17]);
          END;
        END;(*of Damokles *)
      END;(* of ZugNr>0 *)
    END;(* of FOR *)
    wert:=
    gewicht[1]*LONGINT(ZugStat[1]) +gewicht[2]*LONGINT(ZugStat[2])
   +gewicht[3]*LONGINT(ZugStat[3]) +gewicht[4]*LONGINT(ZugStat[4])
   +gewicht[5]*LONGINT(ZugStat[5]) +gewicht[6]*LONGINT(ZugStat[6])
   +gewicht[7]*LONGINT(ZugStat[7])+gewicht[8]*LONGINT(MHLStat[MEMSInd])
   +gewicht[9]*LONGINT(MHLStat[MEMSInd+1])
   +gewicht[10]*LONGINT(MHLStat[MEMSInd+2])
   +gewicht[11]*LONGINT(ZugStat[11]) +gewicht[12]*LONGINT(ZugStat[12])
   +gewicht[13]*LONGINT(ZugStat[13]) +gewicht[14]*LONGINT(ZugStat[14])
   +gewicht[15]*LONGINT(ZugStat[15]) +gewicht[16]*LONGINT(ZugStat[16])
   +gewicht[17]*LONGINT(ZugStat[17])
   +gewicht[18]*LONGINT(MHLStat[YOUMSInd])
   +gewicht[19]*LONGINT(MHLStat[YOUMSInd+1])
   +gewicht[20]*LONGINT(MHLStat[YOUMSInd+2]);
(* WriteLn;
FOR i:=1 TO 17 DO WriteInt(ZugStat[i],3); END;
WriteInt(wert,10);
```

```
WriteLn;
FOR i:=1 TO 8 DO WriteInt(MHLStat[i],3); END;
WriteLn;  *)
RETURN(wert);
END Bewertung;
PROCEDURE Zug(VAR
wert:LONGINT;farbe:Figtype;step:INTEGER;alpha,beta:LONGINT;
      PotZugUnSorted:ARRAY16);
(*  Rekursive Prozedur mit Minimaxverfahren und alfa/beta-Pruning *)
(*  wert: Bewertung des Knotens, an dem Proz. Zug aufgerufen wurde*)
(*  farbe: Farbe des Spielers, der im aufrufenden Knoten am Zug ist*)
(*  step:     Suchtiefe           *)
(*  alfa,beta: Schwellwerte für die alfa/beta-Abbruchbedingungen  *)
(*  PotZugUnSorted: die am aufrufenden Knoten möglichen Folgezüge  *)
VAR  i,j,k,m,n,p :INTEGER; (* Laufvariablen *)
 Gegenfarbe :Figtype;
 zbZug  :INTEGER; (* Position des gerade ausprobierten Zugs   *)
 aktZug  :LONGINT; (* Wert der Stellung,zu der dieser Zug führt*)
 maxZug  :LONGINT; (* Wert der bisher besten Stellung         *)
 YOU,ME,ZKYOU,ZKME:INTEGER;(* Indexoffsets *)
 MEMSInd,YOUMSInd:INTEGER; (* Indexoffsets *)
 ex  :BOOLEAN; (* Flag für Abbruchssteuerung *)
 PotZugSorted  :ARRAY16; (* die vorsortierten möglichen Züge *)
 Increment :INTEGER; (* Incr für MHLStat, abhängig davon, ob
        Diagonalmühle vorliegt oder nicht *)
BEGIN
(* Indexoffsets festlegen *)
   IF farbe=white THEN
      Gegenfarbe:=black;
      YOU:=6;ME:=5;ZKME:=10;ZKYOU:=15;MEMSInd:=1;YOUMSInd:=5;
   ELSE
      Gegenfarbe:=white;
      YOU:=5;ME:=6;ZKME:=15;ZKYOU:=10;MEMSInd:=5;YOUMSInd:=1;
   END;
   IF step>0 THEN
    IF AmZug=farbe THEN maxZug:=-2000000000      (* max-Knoten *)
                  ELSE maxZug:= 2000000000;END; (* min-Knoten *)
      ex:=FALSE;
      SortPotZug(PotZugSorted,PotZugUnSorted);
      i:=0;
(* Züge zu den Folgestellungen ausführen *)
      WHILE i<16 DO
         INC(i);
(* Statusbalken zeichnen *)
         IF StatusInfo THEN
           SetAPen(StatusWinRPPtr,0);
           Move(StatusWinRPPtr,2,10-step);
           Draw(StatusWinRPPtr,146,10-step);
           SetAPen(StatusWinRPPtr,9);
           Move(StatusWinRPPtr,2,10-step);
           Draw(StatusWinRPPtr,2+i*9,10-step);
         END;
         aktZug:=0;
         zbZug:=PotZugSorted[i];
         IF zbZug>0 THEN
(* Board setzen und MHL und ZK aktualisieren *)
        Board[zbZug]:=farbe;
        IF zbZug MOD 4 > 0 THEN
           INC(PotZugSorted[i]);
        ELSE
           PotZugSorted[i]:=0;
        END;
        p:=ZK[zbZug][1]+1;
     FOR m:=2 TO p DO
         n:=ZK[zbZug][m];
     IF (n>56) THEN Increment:=DiagBonus ELSE Increment:=1 END;
     IF (MHL[n][YOU]=3) THEN
        IF (MHL[n][ME]=0) THEN
           IF (step=vardepth) THEN vardepth:=1 END;
           step:=1;
                  END;
               END;
               INC(MHL[n][ME]);
               IF (MHL[n][YOU]=0)  THEN
                  IF MHL[n][ME]=4 THEN
                     ex:=TRUE;
                     IF AmZug=farbe THEN
                        aktZug:=30000000;
                     ELSE
                        aktZug:=-30000000;
                     END;
                  END;
               CASE MHL[n][ME] OF
                     1: INC(MHLStat[MEMSInd],Increment);
                        FOR j:=1 TO 4 DO
                            INC(ZK[MHL[n][j]][ZKME]);
                        END;
                     2: INC(MHLStat[MEMSInd+1],Increment);
                        DEC(MHLStat[MEMSInd],Increment);
                        FOR j:=1 TO 4 DO
                            INC(ZK[MHL[n][j]][ZKME+1]);
                            DEC(ZK[MHL[n][j]][ZKME]);
                        END;
                     3: INC(MHLStat[MEMSInd+2],Increment);
                        DEC(MHLStat[MEMSInd+1],Increment);
                        FOR j:=1 TO 4 DO
                            INC(ZK[MHL[n][j]][ZKME+2]);
                            DEC(ZK[MHL[n][j]][ZKME+1]);
                        END;
                     4: INC(MHLStat[MEMSInd+3],Increment);
                        DEC(MHLStat[MEMSInd+2],Increment);
                        FOR j:=1 TO 4 DO
                            INC(ZK[MHL[n][j]][ZKME+3]);
                            DEC(ZK[MHL[n][j]][ZKME+2]);
                        END;
                  ELSE END;
               END;
               IF (MHL[n][ME]=1) AND (MHL[n][YOU]>0) THEN
                  CASE MHL[n][YOU] OF
                     1: DEC(MHLStat[YOUMSInd],Increment);
                        FOR j:=1 TO 4 DO
                            DEC(ZK[MHL[n][j]][ZKYOU]);
                        END;
                     2: DEC(MHLStat[YOUMSInd+1],Increment);
                        FOR j:=1 TO 4 DO
                            DEC(ZK[MHL[n][j]][ZKYOU+1]);
                        END;
                     3: DEC(MHLStat[YOUMSInd+2],Increment);
                        FOR j:=1 TO 4 DO
                            DEC(ZK[MHL[n][j]][ZKYOU+2]);
                        END;
                     4: DEC(MHLStat[YOUMSInd+3],Increment);
                        FOR j:=1 TO 4 DO
                            DEC(ZK[MHL[n][j]][ZKYOU+3]);
                        END;
                  ELSE END;
               END;
            END;
(* rekursiver Aufruf *)
        IF NOT((aktZug = 30000000) OR (aktZug = -30000000)) THEN
         Zug(aktZug,Gegenfarbe,step-1,alpha,beta,PotZugSorted);
         END;
(* Board zurücksetzen und Änderungen in MHL und ZK zurücknehmen *)
        Board[zbZug]:=pin;
  PotZugSorted[i]:=zbZug;
  FOR m:=2 TO p DO
        n:=ZK[zbZug][m];
        IF (n>56) THEN Increment:=DiagBonus ELSE Increment:=1
END;
                IF (MHL[n][YOU]=0)  THEN
             CASE MHL[n][ME] OF
                     1: DEC(MHLStat[MEMSInd],Increment);
                        FOR j:=1 TO 4 DO
                            DEC(ZK[MHL[n][j]][ZKME]);
                        END;
                     2: DEC(MHLStat[MEMSInd+1],Increment);
                        INC(MHLStat[MEMSInd],Increment);
                        FOR j:=1 TO 4 DO
                            DEC(ZK[MHL[n][j]][ZKME+1]);
                            INC(ZK[MHL[n][j]][ZKME]);
                        END;
                     3: DEC(MHLStat[MEMSInd+2],Increment);
                        INC(MHLStat[MEMSInd+1],Increment);
                        FOR j:=1 TO 4 DO
                            DEC(ZK[MHL[n][j]][ZKME+2]);
                            INC(ZK[MHL[n][j]][ZKME+1]);
                        END;
                     4: DEC(MHLStat[MEMSInd+3],Increment);
                        INC(MHLStat[MEMSInd+2],Increment);
                        FOR j:=1 TO 4 DO
                            DEC(ZK[MHL[n][j]][ZKME+3]);
                            INC(ZK[MHL[n][j]][ZKME+2]);
                        END;
                  ELSE END;
               END;
               IF (MHL[n][ME]=1) AND (MHL[n][YOU]>0) THEN
                  CASE MHL[n][YOU] OF
                     1: INC(MHLStat[YOUMSInd],Increment);
                        FOR j:=1 TO 4 DO
                            INC(ZK[MHL[n][j]][ZKYOU]);
                        END;
                     2: INC(MHLStat[YOUMSInd+1],Increment);
                        FOR j:=1 TO 4 DO
                            INC(ZK[MHL[n][j]][ZKYOU+1]);
                        END;
                     3: INC(MHLStat[YOUMSInd+2],Increment);
                        FOR j:=1 TO 4 DO
                            INC(ZK[MHL[n][j]][ZKYOU+2]);
                        END;
                     4: INC(MHLStat[YOUMSInd+3],Increment);
                        FOR j:=1 TO 4 DO
                            INC(ZK[MHL[n][j]][ZKYOU+3]);
                        END;
                  ELSE END;
               END;
                DEC(MHL[n][ME]);
            END;
(* Minimaxverfahren für max-Knoten *)
            IF (AmZug = farbe) THEN
               IF (aktZug > maxZug) THEN
          maxZug:=aktZug;
      IF (step = vardepth) THEN
         BestZug:=zbZug;
      END;
         END;
         IF (aktZug = maxZug) AND (RND(2) = 1) THEN
      maxZug:=aktZug;
      IF (step = vardepth) THEN
         BestZug:=zbZug;
      END;
         END;          (* beta-Pruning an max-Knoten *)
         IF (maxZug > alpha)  THEN alpha:=maxZug;END;
         IF (step # vardepth) AND (maxZug > beta) THEN
            betaAnz:=betaAnz+1;
                  IF StatusInfo THEN
              ValToStr(betaAnz,FALSE,S10,10,8," ",err);
                    Move(StatusWinRPPtr,80,65);
                    Text(StatusWinRPPtr,ADR(S10),Length(S10));
                  END;
            i:=17;
         END;
        ELSE
(* Minimaxverfahren an min-Knoten *)
         IF (aktZug < maxZug) THEN
        maxZug:=aktZug;
      IF (step = vardepth) THEN
         BestZug:=zbZug;
      END;
         END;
         IF (aktZug = maxZug) AND (RND(2) = 1) THEN
      maxZug:=aktZug;
      IF (step = vardepth) THEN
         BestZug:=zbZug;
      END;
         END;
(* alfa-Pruning an min-Knoten *)
         IF (maxZug < beta)  THEN beta:=maxZug;END;
         IF (step # vardepth) AND (maxZug < alpha) THEN
            alphaAnz:=alphaAnz+1;
            IF StatusInfo THEN
               ValToStr(alphaAnz,FALSE,S10,10,8," ",err);
                    Move(StatusWinRPPtr,80,55);

Text(StatusWinRPPtr,ADR(S10),Length(S10));
                  END;
            i:=17;
         END;
        END;
   END;(* of IF *)
        IF (SpielWindowPtr # NIL) THEN
           IntuiMsg:=GetMsg(SpielWindowPtr^.userPort);
           IF IntuiMsg # NIL THEN
              class:=IntuiMsg^.class;
              code :=IntuiMsg^.code;
              ReplyMsg(IntuiMsg);
              interrupt := TRUE;
           END;
        END;
```

```
        IF ex OR interrupt THEN
      i:=17;
   END;
       END;(* of WHILE *)
    ELSE    (* Bewertung der willkürlichen Endstellungen *)
       IF AmZug=farbe THEN maxZug:=Bewertung(farbe,PotZugUnSorted);
                    ELSE maxZug:=-Bewertung(farbe,PotZugUnSorted);
       END;
    END;
    wert:=maxZug;
END Zug;
PROCEDURE InitStrat;   (* initialisiert die Variablen   *)
VAR i,j : INTEGER;
BEGIN
  Verl:=pin;
  interrupt:=FALSE;
FOR i:=1 TO 64 DO
  ZugListe[i]:=0;
  FOR j:=9 TO 10 DO
    ZK[i][j]:=0;
  END;
END;
ZLIndex:=0;
FOR i:=1 TO 64 DO
  FOR j:=9 TO 20 DO
   ZK[i][j]:=0;
  END;
END;
FOR i:=1 TO 8 DO MHLStat[i]:=0;END;
ZK[ 1][ 1]:=7; ZK[ 1][ 2]:=1; ZK[ 1][ 3]:=17; ZK[ 1][ 4]:=33;
ZK[ 1][ 5]:=49; ZK[ 1][ 6]:=57; ZK[ 1][ 7]:=65; ZK[ 1][ 8]:=73;
ZK[ 2][ 1]:=4; ZK[ 2][ 2]:=1; ZK[ 2][ 3]:=18; ZK[ 2][ 4]:=34;
ZK[ 2][ 5]:=50; ZK[ 2][ 6]:=0; ZK[ 2][ 7]:=0; ZK[ 2][ 8]:=0;
ZK[ 3][ 1]:=4; ZK[ 3][ 2]:=1; ZK[ 3][ 3]:=19; ZK[ 3][ 4]:=35;
ZK[ 3][ 5]:=51; ZK[ 3][ 6]:=0; ZK[ 3][ 7]:=0; ZK[ 3][ 8]:=0;
ZK[ 4][ 1]:=7; ZK[ 4][ 2]:=1; ZK[ 4][ 3]:=20; ZK[ 4][ 4]:=36;
ZK[ 4][ 5]:=52; ZK[ 4][ 6]:=58; ZK[ 4][ 7]:=66; ZK[ 4][ 8]:=74;
ZK[ 5][ 1]:=4; ZK[ 5][ 2]:=2; ZK[ 5][ 3]:=17; ZK[ 5][ 4]:=37;
ZK[ 5][ 5]:=67; ZK[ 5][ 6]:=0; ZK[ 5][ 7]:=0; ZK[ 5][ 8]:=0;
ZK[ 6][ 1]:=4; ZK[ 6][ 2]:=2; ZK[ 6][ 3]:=18; ZK[ 6][ 4]:=38;
ZK[ 6][ 5]:=57; ZK[ 6][ 6]:=0; ZK[ 6][ 7]:=0; ZK[ 6][ 8]:=0;
ZK[ 7][ 1]:=4; ZK[ 7][ 2]:=2; ZK[ 7][ 3]:=19; ZK[ 7][ 4]:=39;
ZK[ 7][ 5]:=58; ZK[ 7][ 6]:=0; ZK[ 7][ 7]:=0; ZK[ 7][ 8]:=0;
ZK[ 8][ 1]:=4; ZK[ 8][ 2]:=2; ZK[ 8][ 3]:=20; ZK[ 8][ 4]:=40;
ZK[ 8][ 5]:=68; ZK[ 8][ 6]:=0; ZK[ 8][ 7]:=0; ZK[ 8][ 8]:=0;
ZK[ 9][ 1]:=4; ZK[ 9][ 2]:=3; ZK[ 9][ 3]:=17; ZK[ 9][ 4]:=41;
ZK[ 9][ 5]:=69; ZK[ 9][ 6]:=0; ZK[ 9][ 7]:=0; ZK[ 9][ 8]:=0;
ZK[10][ 1]:=4; ZK[10][ 2]:=3; ZK[10][ 3]:=18; ZK[10][ 4]:=42;
ZK[10][ 5]:=58; ZK[10][ 6]:=0; ZK[10][ 7]:=0; ZK[10][ 8]:=0;
ZK[11][ 1]:=4; ZK[11][ 2]:=3; ZK[11][ 3]:=19; ZK[11][ 4]:=43;
ZK[11][ 5]:=57; ZK[11][ 6]:=0; ZK[11][ 7]:=0; ZK[11][ 8]:=0;
ZK[12][ 1]:=4; ZK[12][ 2]:=3; ZK[12][ 3]:=20; ZK[12][ 4]:=44;
ZK[12][ 5]:=70; ZK[12][ 6]:=0; ZK[12][ 7]:=0; ZK[12][ 8]:=0;
ZK[13][ 1]:=7; ZK[13][ 2]:=4; ZK[13][ 3]:=17; ZK[13][ 4]:=45;
ZK[13][ 5]:=53; ZK[13][ 6]:=58; ZK[13][ 7]:=71; ZK[13][ 8]:=75;
ZK[14][ 1]:=4; ZK[14][ 2]:=4; ZK[14][ 3]:=18; ZK[14][ 4]:=46;
ZK[14][ 5]:=54; ZK[14][ 6]:=0; ZK[14][ 7]:=0; ZK[14][ 8]:=0;
ZK[15][ 1]:=4; ZK[15][ 2]:=4; ZK[15][ 3]:=19; ZK[15][ 4]:=47;
ZK[15][ 5]:=55; ZK[15][ 6]:=0; ZK[15][ 7]:=0; ZK[15][ 8]:=0;
ZK[16][ 1]:=7; ZK[16][ 2]:=4; ZK[16][ 3]:=20; ZK[16][ 4]:=48;
ZK[16][ 5]:=56; ZK[16][ 6]:=57; ZK[16][ 7]:=72; ZK[16][ 8]:=76;
ZK[17][ 1]:=4; ZK[17][ 2]:=5; ZK[17][ 3]:=21; ZK[17][ 4]:=33;
ZK[17][ 5]:=59; ZK[17][ 6]:=0; ZK[17][ 7]:=0; ZK[17][ 8]:=0;
ZK[18][ 1]:=4; ZK[18][ 2]:=5; ZK[18][ 3]:=22; ZK[18][ 4]:=34;
ZK[18][ 5]:=65; ZK[18][ 6]:=0; ZK[18][ 7]:=0; ZK[18][ 8]:=0;
ZK[19][ 1]:=4; ZK[19][ 2]:=5; ZK[19][ 3]:=23; ZK[19][ 4]:=35;
ZK[19][ 5]:=66; ZK[19][ 6]:=0; ZK[19][ 7]:=0; ZK[19][ 8]:=0;
ZK[20][ 1]:=4; ZK[20][ 2]:=5; ZK[20][ 3]:=24; ZK[20][ 4]:=36;
ZK[20][ 5]:=60; ZK[20][ 6]:=0; ZK[20][ 7]:=0; ZK[20][ 8]:=0;
ZK[21][ 1]:=4; ZK[21][ 2]:=6; ZK[21][ 3]:=21; ZK[21][ 4]:=37;
ZK[21][ 5]:=49; ZK[21][ 6]:=0; ZK[21][ 7]:=0; ZK[21][ 8]:=0;
ZK[22][ 1]:=7; ZK[22][ 2]:=6; ZK[22][ 3]:=22; ZK[22][ 4]:=38;
ZK[22][ 5]:=50; ZK[22][ 6]:=59; ZK[22][ 7]:=67; ZK[22][ 8]:=73;
ZK[23][ 1]:=7; ZK[23][ 2]:=6; ZK[23][ 3]:=23; ZK[23][ 4]:=39;
ZK[23][ 5]:=51; ZK[23][ 6]:=60; ZK[23][ 7]:=68; ZK[23][ 8]:=74;
ZK[24][ 1]:=4; ZK[24][ 2]:=6; ZK[24][ 3]:=24; ZK[24][ 4]:=40;
ZK[24][ 5]:=52; ZK[24][ 6]:=0; ZK[24][ 7]:=0; ZK[24][ 8]:=0;
ZK[25][ 1]:=4; ZK[25][ 2]:=7; ZK[25][ 3]:=21; ZK[25][ 4]:=41;
ZK[25][ 5]:=53; ZK[25][ 6]:=0; ZK[25][ 7]:=0; ZK[25][ 8]:=0;
ZK[26][ 1]:=7; ZK[26][ 2]:=7; ZK[26][ 3]:=22; ZK[26][ 4]:=42;
ZK[26][ 5]:=54; ZK[26][ 6]:=60; ZK[26][ 7]:=69; ZK[26][ 8]:=75;
ZK[27][ 1]:=7; ZK[27][ 2]:=7; ZK[27][ 3]:=23; ZK[27][ 4]:=43;
ZK[27][ 5]:=55; ZK[27][ 6]:=59; ZK[27][ 7]:=70; ZK[27][ 8]:=76;
ZK[28][ 1]:=4; ZK[28][ 2]:=7; ZK[28][ 3]:=24; ZK[28][ 4]:=44;
ZK[28][ 5]:=56; ZK[28][ 6]:=0; ZK[28][ 7]:=0; ZK[28][ 8]:=0;
ZK[29][ 1]:=4; ZK[29][ 2]:=8; ZK[29][ 3]:=21; ZK[29][ 4]:=45;
ZK[29][ 5]:=60; ZK[29][ 6]:=0; ZK[29][ 7]:=0; ZK[29][ 8]:=0;
ZK[30][ 1]:=4; ZK[30][ 2]:=8; ZK[30][ 3]:=22; ZK[30][ 4]:=46;
ZK[30][ 5]:=71; ZK[30][ 6]:=0; ZK[30][ 7]:=0; ZK[30][ 8]:=0;
ZK[31][ 1]:=4; ZK[31][ 2]:=8; ZK[31][ 3]:=23; ZK[31][ 4]:=47;
ZK[31][ 5]:=72; ZK[31][ 6]:=0; ZK[31][ 7]:=0; ZK[31][ 8]:=0;
ZK[32][ 1]:=4; ZK[32][ 2]:=8; ZK[32][ 3]:=24; ZK[32][ 4]:=48;
ZK[32][ 5]:=59; ZK[32][ 6]:=0; ZK[32][ 7]:=0; ZK[32][ 8]:=0;
ZK[33][ 1]:=4; ZK[33][ 2]:=9; ZK[33][ 3]:=25; ZK[33][ 4]:=33;
ZK[33][ 5]:=61; ZK[33][ 6]:=0; ZK[33][ 7]:=0; ZK[33][ 8]:=0;
ZK[34][ 1]:=4; ZK[34][ 2]:=9; ZK[34][ 3]:=26; ZK[34][ 4]:=34;
ZK[34][ 5]:=66; ZK[34][ 6]:=0; ZK[34][ 7]:=0; ZK[34][ 8]:=0;
ZK[35][ 1]:=4; ZK[35][ 2]:=9; ZK[35][ 3]:=27; ZK[35][ 4]:=35;
ZK[35][ 5]:=65; ZK[35][ 6]:=0; ZK[35][ 7]:=0; ZK[35][ 8]:=0;
ZK[36][ 1]:=4; ZK[36][ 2]:=9; ZK[36][ 3]:=28; ZK[36][ 4]:=36;
ZK[36][ 5]:=62; ZK[36][ 6]:=0; ZK[36][ 7]:=0; ZK[36][ 8]:=0;
ZK[37][ 1]:=4; ZK[37][ 2]:=10; ZK[37][ 3]:=25; ZK[37][ 4]:=37;
ZK[37][ 5]:=53; ZK[37][ 6]:=0; ZK[37][ 7]:=0; ZK[37][ 8]:=0;
ZK[38][ 1]:=7; ZK[38][ 2]:=10; ZK[38][ 3]:=26; ZK[38][ 4]:=38;
ZK[38][ 5]:=54; ZK[38][ 6]:=61; ZK[38][ 7]:=68; ZK[38][ 8]:=76;
ZK[39][ 1]:=7; ZK[39][ 2]:=10; ZK[39][ 3]:=27; ZK[39][ 4]:=39;
ZK[39][ 5]:=55; ZK[39][ 6]:=62; ZK[39][ 7]:=67; ZK[39][ 8]:=75;
ZK[40][ 1]:=4; ZK[40][ 2]:=10; ZK[40][ 3]:=28; ZK[40][ 4]:=40;
ZK[40][ 5]:=56; ZK[40][ 6]:=0; ZK[40][ 7]:=0; ZK[40][ 8]:=0;
ZK[41][ 1]:=4; ZK[41][ 2]:=11; ZK[41][ 3]:=25; ZK[41][ 4]:=41;
ZK[41][ 5]:=49; ZK[41][ 6]:=0; ZK[41][ 7]:=0; ZK[41][ 8]:=0;
ZK[42][ 1]:=7; ZK[42][ 2]:=11; ZK[42][ 3]:=26; ZK[42][ 4]:=42;
ZK[42][ 5]:=50; ZK[42][ 6]:=62; ZK[42][ 7]:=70; ZK[42][ 8]:=74;
ZK[43][ 1]:=7; ZK[43][ 2]:=11; ZK[43][ 3]:=27; ZK[43][ 4]:=43;
ZK[43][ 5]:=51; ZK[43][ 6]:=61; ZK[43][ 7]:=69; ZK[43][ 8]:=73;
ZK[44][ 1]:=4; ZK[44][ 2]:=11; ZK[44][ 3]:=28; ZK[44][ 4]:=44;
ZK[44][ 5]:=52; ZK[44][ 6]:=0; ZK[44][ 7]:=0; ZK[44][ 8]:=0;
ZK[45][ 1]:=4; ZK[45][ 2]:=12; ZK[45][ 3]:=25; ZK[45][ 4]:=45;
ZK[45][ 5]:=62; ZK[45][ 6]:=0; ZK[45][ 7]:=0; ZK[45][ 8]:=0;
ZK[46][ 1]:=4; ZK[46][ 2]:=12; ZK[46][ 3]:=26; ZK[46][ 4]:=46;
ZK[46][ 5]:=72; ZK[46][ 6]:=0; ZK[46][ 7]:=0; ZK[46][ 8]:=0;
ZK[47][ 1]:=4; ZK[47][ 2]:=12; ZK[47][ 3]:=27; ZK[47][ 4]:=47;
ZK[47][ 5]:=71; ZK[47][ 6]:=0; ZK[47][ 7]:=0; ZK[47][ 8]:=0;
ZK[48][ 1]:=4; ZK[48][ 2]:=12; ZK[48][ 3]:=28; ZK[48][ 4]:=48;
ZK[48][ 5]:=61; ZK[48][ 6]:=0; ZK[48][ 7]:=0; ZK[48][ 8]:=0;
ZK[49][ 1]:=7; ZK[49][ 2]:=13; ZK[49][ 3]:=29; ZK[49][ 4]:=33;
ZK[49][ 5]:=53; ZK[49][ 6]:=63; ZK[49][ 7]:=66; ZK[49][ 8]:=76;
ZK[50][ 1]:=4; ZK[50][ 2]:=13; ZK[50][ 3]:=30; ZK[50][ 4]:=34;
ZK[50][ 5]:=54; ZK[50][ 6]:=0; ZK[50][ 7]:=0; ZK[50][ 8]:=0;
ZK[51][ 1]:=4; ZK[51][ 2]:=13; ZK[51][ 3]:=31; ZK[51][ 4]:=35;
ZK[51][ 5]:=55; ZK[51][ 6]:=0; ZK[51][ 7]:=0; ZK[51][ 8]:=0;
ZK[52][ 1]:=7; ZK[52][ 2]:=13; ZK[52][ 3]:=32; ZK[52][ 4]:=36;
ZK[52][ 5]:=56; ZK[52][ 6]:=64; ZK[52][ 7]:=65; ZK[52][ 8]:=75;
ZK[53][ 1]:=4; ZK[53][ 2]:=14; ZK[53][ 3]:=29; ZK[53][ 4]:=37;
ZK[53][ 5]:=68; ZK[53][ 6]:=0; ZK[53][ 7]:=0; ZK[53][ 8]:=0;
ZK[54][ 1]:=4; ZK[54][ 2]:=14; ZK[54][ 3]:=30; ZK[54][ 4]:=38;
ZK[54][ 5]:=63; ZK[54][ 6]:=0; ZK[54][ 7]:=0; ZK[54][ 8]:=0;
ZK[55][ 1]:=4; ZK[55][ 2]:=14; ZK[55][ 3]:=31; ZK[55][ 4]:=39;
ZK[55][ 5]:=64; ZK[55][ 6]:=0; ZK[55][ 7]:=0; ZK[55][ 8]:=0;
ZK[56][ 1]:=4; ZK[56][ 2]:=14; ZK[56][ 3]:=32; ZK[56][ 4]:=40;
ZK[56][ 5]:=67; ZK[56][ 6]:=0; ZK[56][ 7]:=0; ZK[56][ 8]:=0;
ZK[57][ 1]:=4; ZK[57][ 2]:=15; ZK[57][ 3]:=29; ZK[57][ 4]:=41;
ZK[57][ 5]:=70; ZK[57][ 6]:=0; ZK[57][ 7]:=0; ZK[57][ 8]:=0;
ZK[58][ 1]:=4; ZK[58][ 2]:=15; ZK[58][ 3]:=30; ZK[58][ 4]:=42;
ZK[58][ 5]:=64; ZK[58][ 6]:=0; ZK[58][ 7]:=0; ZK[58][ 8]:=0;
ZK[59][ 1]:=4; ZK[59][ 2]:=15; ZK[59][ 3]:=31; ZK[59][ 4]:=43;
ZK[59][ 5]:=63; ZK[59][ 6]:=0; ZK[59][ 7]:=0; ZK[59][ 8]:=0;
ZK[60][ 1]:=4; ZK[60][ 2]:=15; ZK[60][ 3]:=32; ZK[60][ 4]:=44;
ZK[60][ 5]:=69; ZK[60][ 6]:=0; ZK[60][ 7]:=0; ZK[60][ 8]:=0;
ZK[61][ 1]:=7; ZK[61][ 2]:=16; ZK[61][ 3]:=29; ZK[61][ 4]:=45;
ZK[61][ 5]:=49; ZK[61][ 6]:=64; ZK[61][ 7]:=72; ZK[61][ 8]:=74;
ZK[62][ 1]:=4; ZK[62][ 2]:=16; ZK[62][ 3]:=30; ZK[62][ 4]:=46;
ZK[62][ 5]:=50; ZK[62][ 6]:=0; ZK[62][ 7]:=0; ZK[62][ 8]:=0;
ZK[63][ 1]:=4; ZK[63][ 2]:=16; ZK[63][ 3]:=31; ZK[63][ 4]:=47;
ZK[63][ 5]:=51; ZK[63][ 6]:=0; ZK[63][ 7]:=0; ZK[63][ 8]:=0;
ZK[64][ 1]:=7; ZK[64][ 2]:=16; ZK[64][ 3]:=32; ZK[64][ 4]:=48;
ZK[64][ 5]:=52; ZK[64][ 6]:=63; ZK[64][ 7]:=71; ZK[64][ 8]:=73;
FOR i:=1 TO 76 DO FOR j:=5 TO 6 DO MHL[i][j]:=0;END;END;
(* senkrechte Mühlen *)
MHL[ 1][ 1]:=1; MHL[ 1][ 2]:=2; MHL[ 1][ 3]:=3; MHL[ 1][ 4]:=4;
MHL[ 2][ 1]:=5; MHL[ 2][ 2]:=6; MHL[ 2][ 3]:=7; MHL[ 2][ 4]:=8;
MHL[ 3][ 1]:=9; MHL[ 3][ 2]:=10; MHL[ 3][ 3]:=11; MHL[ 3][ 4]:=12;
MHL[ 4][ 1]:=13; MHL[ 4][ 2]:=14; MHL[ 4][ 3]:=15; MHL[ 4][ 4]:=16;
MHL[ 5][ 1]:=17; MHL[ 5][ 2]:=18; MHL[ 5][ 3]:=19; MHL[ 5][ 4]:=20;
MHL[ 6][ 1]:=21; MHL[ 6][ 2]:=22; MHL[ 6][ 3]:=23; MHL[ 6][ 4]:=24;
MHL[ 7][ 1]:=25; MHL[ 7][ 2]:=26; MHL[ 7][ 3]:=27; MHL[ 7][ 4]:=28;
MHL[ 8][ 1]:=29; MHL[ 8][ 2]:=30; MHL[ 8][ 3]:=31; MHL[ 8][ 4]:=32;
MHL[ 9][ 1]:=33; MHL[ 9][ 2]:=34; MHL[ 9][ 3]:=35; MHL[ 9][ 4]:=36;
MHL[10][ 1]:=37; MHL[10][ 2]:=38; MHL[10][ 3]:=39; MHL[10][ 4]:=40;
MHL[11][ 1]:=41; MHL[11][ 2]:=42; MHL[11][ 3]:=43; MHL[11][ 4]:=44;
MHL[12][ 1]:=45; MHL[12][ 2]:=46; MHL[12][ 3]:=47; MHL[12][ 4]:=48;
MHL[13][ 1]:=49; MHL[13][ 2]:=50; MHL[13][ 3]:=51; MHL[13][ 4]:=52;
MHL[14][ 1]:=53; MHL[14][ 2]:=54; MHL[14][ 3]:=55; MHL[14][ 4]:=56;
MHL[15][ 1]:=57; MHL[15][ 2]:=58; MHL[15][ 3]:=59; MHL[15][ 4]:=60;
MHL[16][ 1]:=61; MHL[16][ 2]:=62; MHL[16][ 3]:=63; MHL[16][ 4]:=64;
(* waagerechte Mühlen *)
MHL[17][ 1]:=1; MHL[17][ 2]:=5; MHL[17][ 3]:=9; MHL[17][ 4]:=13;
MHL[18][ 1]:=2; MHL[18][ 2]:=6; MHL[18][ 3]:=10; MHL[18][ 4]:=14;
MHL[19][ 1]:=3; MHL[19][ 2]:=7; MHL[19][ 3]:=11; MHL[19][ 4]:=15;
MHL[20][ 1]:=4; MHL[20][ 2]:=8; MHL[20][ 3]:=12; MHL[20][ 4]:=16;
MHL[21][ 1]:=17; MHL[21][ 2]:=21; MHL[21][ 3]:=25; MHL[21][ 4]:=29;
MHL[22][ 1]:=18; MHL[22][ 2]:=22; MHL[22][ 3]:=26; MHL[22][ 4]:=30;
MHL[23][ 1]:=19; MHL[23][ 2]:=23; MHL[23][ 3]:=27; MHL[23][ 4]:=31;
MHL[24][ 1]:=20; MHL[24][ 2]:=24; MHL[24][ 3]:=28; MHL[24][ 4]:=32;
MHL[25][ 1]:=33; MHL[25][ 2]:=37; MHL[25][ 3]:=41; MHL[25][ 4]:=45;
MHL[26][ 1]:=34; MHL[26][ 2]:=38; MHL[26][ 3]:=42; MHL[26][ 4]:=46;
MHL[27][ 1]:=35; MHL[27][ 2]:=39; MHL[27][ 3]:=43; MHL[27][ 4]:=47;
MHL[28][ 1]:=36; MHL[28][ 2]:=40; MHL[28][ 3]:=44; MHL[28][ 4]:=48;
MHL[29][ 1]:=49; MHL[29][ 2]:=53; MHL[29][ 3]:=57; MHL[29][ 4]:=61;
MHL[30][ 1]:=50; MHL[30][ 2]:=54; MHL[30][ 3]:=58; MHL[30][ 4]:=62;
MHL[31][ 1]:=51; MHL[31][ 2]:=55; MHL[31][ 3]:=59; MHL[31][ 4]:=63;
MHL[32][ 1]:=52; MHL[32][ 2]:=56; MHL[32][ 3]:=60; MHL[32][ 4]:=64;
MHL[33][ 1]:=1; MHL[33][ 2]:=17; MHL[33][ 3]:=33; MHL[33][ 4]:=49;
MHL[34][ 1]:=2; MHL[34][ 2]:=18; MHL[34][ 3]:=34; MHL[34][ 4]:=50;
MHL[35][ 1]:=3; MHL[35][ 2]:=19; MHL[35][ 3]:=35; MHL[35][ 4]:=51;
MHL[36][ 1]:=4; MHL[36][ 2]:=20; MHL[36][ 3]:=36; MHL[36][ 4]:=52;
MHL[37][ 1]:=5; MHL[37][ 2]:=21; MHL[37][ 3]:=37; MHL[37][ 4]:=53;
MHL[38][ 1]:=6; MHL[38][ 2]:=22; MHL[38][ 3]:=38; MHL[38][ 4]:=54;
MHL[39][ 1]:=7; MHL[39][ 2]:=23; MHL[39][ 3]:=39; MHL[39][ 4]:=55;
MHL[40][ 1]:=8; MHL[40][ 2]:=24; MHL[40][ 3]:=40; MHL[40][ 4]:=56;
MHL[41][ 1]:=9; MHL[41][ 2]:=25; MHL[41][ 3]:=41; MHL[41][ 4]:=57;
MHL[42][ 1]:=10; MHL[42][ 2]:=26; MHL[42][ 3]:=42; MHL[42][ 4]:=58;
MHL[43][ 1]:=11; MHL[43][ 2]:=27; MHL[43][ 3]:=43; MHL[43][ 4]:=59;
MHL[44][ 1]:=12; MHL[44][ 2]:=28; MHL[44][ 3]:=44; MHL[44][ 4]:=60;
MHL[45][ 1]:=13; MHL[45][ 2]:=29; MHL[45][ 3]:=45; MHL[45][ 4]:=61;
MHL[46][ 1]:=14; MHL[46][ 2]:=30; MHL[46][ 3]:=46; MHL[46][ 4]:=62;
MHL[47][ 1]:=15; MHL[47][ 2]:=31; MHL[47][ 3]:=47; MHL[47][ 4]:=63;
MHL[48][ 1]:=16; MHL[48][ 2]:=32; MHL[48][ 3]:=48; MHL[48][ 4]:=64;
MHL[49][ 1]:=1; MHL[49][ 2]:=21; MHL[49][ 3]:=41; MHL[49][ 4]:=61;
MHL[50][ 1]:=2; MHL[50][ 2]:=22; MHL[50][ 3]:=42; MHL[50][ 4]:=62;
MHL[51][ 1]:=3; MHL[51][ 2]:=23; MHL[51][ 3]:=43; MHL[51][ 4]:=63;
MHL[52][ 1]:=4; MHL[52][ 2]:=24; MHL[52][ 3]:=44; MHL[52][ 4]:=64;
MHL[53][ 1]:=13; MHL[53][ 2]:=25; MHL[53][ 3]:=37; MHL[53][ 4]:=49;
MHL[54][ 1]:=14; MHL[54][ 2]:=26; MHL[54][ 3]:=38; MHL[54][ 4]:=50;
MHL[55][ 1]:=15; MHL[55][ 2]:=27; MHL[55][ 3]:=39; MHL[55][ 4]:=51;
MHL[56][ 1]:=16; MHL[56][ 2]:=28; MHL[56][ 3]:=40; MHL[56][ 4]:=52;
(* diagonale Mühlen *)
MHL[57][ 1]:=1; MHL[57][ 2]:=6; MHL[57][ 3]:=11; MHL[57][ 4]:=16;
MHL[58][ 1]:=4; MHL[58][ 2]:=7; MHL[58][ 3]:=10; MHL[58][ 4]:=13;
MHL[59][ 1]:=17; MHL[59][ 2]:=22; MHL[59][ 3]:=27; MHL[59][ 4]:=32;
MHL[60][ 1]:=20; MHL[60][ 2]:=23; MHL[60][ 3]:=26; MHL[60][ 4]:=29;
MHL[61][ 1]:=33; MHL[61][ 2]:=38; MHL[61][ 3]:=43; MHL[61][ 4]:=48;
MHL[62][ 1]:=36; MHL[62][ 2]:=39; MHL[62][ 3]:=42; MHL[62][ 4]:=45;
MHL[63][ 1]:=49; MHL[63][ 2]:=54; MHL[63][ 3]:=59; MHL[63][ 4]:=64;
MHL[64][ 1]:=52; MHL[64][ 2]:=55; MHL[64][ 3]:=58; MHL[64][ 4]:=61;
MHL[65][ 1]:=1; MHL[65][ 2]:=18; MHL[65][ 3]:=35; MHL[65][ 4]:=52;
MHL[66][ 1]:=4; MHL[66][ 2]:=19; MHL[66][ 3]:=34; MHL[66][ 4]:=49;
MHL[67][ 1]:=5; MHL[67][ 2]:=22; MHL[67][ 3]:=39; MHL[67][ 4]:=56;
MHL[68][ 1]:=8; MHL[68][ 2]:=23; MHL[68][ 3]:=38; MHL[68][ 4]:=53;
MHL[69][ 1]:=9; MHL[69][ 2]:=26; MHL[69][ 3]:=43; MHL[69][ 4]:=60;
MHL[70][ 1]:=12; MHL[70][ 2]:=27; MHL[70][ 3]:=42; MHL[70][ 4]:=57;
MHL[71][ 1]:=13; MHL[71][ 2]:=30; MHL[71][ 3]:=47; MHL[71][ 4]:=64;
MHL[72][ 1]:=16; MHL[72][ 2]:=31; MHL[72][ 3]:=46; MHL[72][ 4]:=61;
(* raumdiagonale Mühlen *)
MHL[73][ 1]:=1; MHL[73][ 2]:=22; MHL[73][ 3]:=43; MHL[73][ 4]:=64;
MHL[74][ 1]:=4; MHL[74][ 2]:=23; MHL[74][ 3]:=42; MHL[74][ 4]:=61;
MHL[75][ 1]:=13; MHL[75][ 2]:=26; MHL[75][ 3]:=39; MHL[75][ 4]:=52;
MHL[76][ 1]:=16; MHL[76][ 2]:=27; MHL[76][ 3]:=38; MHL[76][ 4]:=49;
FOR i:=1 TO 64 DO Board[i]:=pin;END;
SetStatusInfo(1);
SetDepth(1);
END InitStrat;
END sogostrat.                    © 1993 M&T
```

»Listing 4: Das Implementationsmodul des »Strategiekerns« unseres Spiels

Libraries für BASIC

Selbstgestrickt

Wie programmiert man eigentlich eine Library? Raphael Koch aus Vechelde beschreibt, wie man´s macht und wie Sie Ihre eigene Funktionsbibliothek dann von BASIC aus nutzen.

von Raphael Koch

Das Betriebssystem des Amiga besteht zu einem sehr großen Teil aus den Libraries. Das ist auch sehr praktisch, denn eine schon bestehende Library kann man recht einfach um Befehle und Funktionen erweitern. Zur Erinnerung: Befehle und Funktionen unterscheiden sich dadurch, daß eine Funktion einen Wert als Returncode im Register d0 liefert und ein Befehl nicht.

Aber das Programmieren einer – eigenen – neuen Library, die von Diskette nachgeladen wird, ist schon eine Wissenschaft für sich. Um Ihnen das Ganze einfacher zu machen, zeigen wir Ihnen hier, wie so eine Library erstellt wird.

Wenn das Betriebssystem eine Library von der Diskette nachlädt, werden eine ganze Menge Routinen intern vom Betriebssystem aufgerufen, um die Library im Speicher zu initialisieren. Wer mehr über die Theorie der Initialisierung erfahren möchte, sei auf den siebten Teil des Insiderkurses im AMIGA Magazin, Ausgabe 11, im Jahr 1989 verwiesen. Dazu werden eine ganze Menge Daten benötigt, die wir in unserem Programm festlegen müssen (siehe Listings Seite 46).

Da das Betriebssystem Libraries wie Resident-Module behandelt, schreiben wir als erstes eine Resident-Struktur in unser Programm. Diese sieht wie der Tabelle unten gezeigt aus:

Offset (Dez.)	Größe	Bedeutung
0	word	Kennung für Resident-Struktur dez.:19196 hex:4AFC
2	aptr	Zeiger auf Anfang der Struktur
6	aptr	Zeiger auf Programmende
10	byte	Flags: wir tragen hier 128 ein
11	byte	für Eintrag Library-Version
12	byte	Type: für Library 9 eingetragen
13	byte	Die Priorität: normal ist 0
14	aptr	Zeiger auf Namen der Library
18	aptr	Zeiger auf Identifikationsstring
22	aptr	-> Anfang der LibInit-Struktur

In einer Resident-Struktur muß auf jeden Fall als allererstes ein Identifikationscode stehen. Hier lautet er 19196. In unser Library steht also als erstes nach den Konstantenzuweisungen und den eventuellen Include-Befehlen die Zeile

```
dc.w 19196
```

oder

```
dc.w $4AFC
```

Oder, wer es lieber mag, kann auch

```
illegal
```

eintragen. Und damit das Betriebssystem ganz sicher sein kann, daß es sich um ein Resident-Modul handelt, muß als nächstes ein Zeiger auf den Beginn des Resident-Moduls stehen. Also steht im Programm:

```
ResidentStruct dc.w 19196
               dc.l ResidentStruct
```

Wer seine Library ganz narrensicher gestalten will, kann vor der Resident-Struktur auch noch

```
moveq #20,d0
rts
```

ins Programm schreiben, um bei einem eventuellem Start vom CLI durch einen unerfahrenen Anwender einen Absturz zu verhindern. Aber jetzt weiter in der Resident-Struktur: Nach den Sicherheitszeilen muß dem System noch mitgeteilt werden, bis wohin unser Resident-Modul, also die Library, im Speicher reicht. Dazu schreiben wir hinter den letzten Befehl und hinter die letzte Assemblerdirektive eine Sprungmarke. Diese Marke tragen wir als nächstes in unsere Struktur ein. Die weiteren Einträge dürften sich selbst erklären.
Die folgende Tabelle zeigt die »InitLib«-Struktur:

Offset (Dez.)	Bedeutung
0	Die Größe der Basis
4	Zeiger auf die Sprungtabelle
8	Zeiger auf eine Datentabelle
12	Zeiger auf die InitLib Routine

Bei der Datentabelle handelt es sich um Daten, mit denen die Basis unserer Library initialisiert wird. (Da wir auf Include-Files, in Hinsicht auf diejenigen, die keine benutzen können, verzichten möchten, werden die benötigten Makros am Anfang unserer Beispiel-Library beschrieben. Die Include-Files, die wir dennoch genutzt haben, können Sie mit dem Programm »MakeInclude« selbst aus den lib.fd's für BASIC generieren.) Die Tabelle wird durch das Langwort Null (0) abgeschlossen.Im allgemeinen wird folgendes eingetragen:

```
DataTable INITBYTE 8,9 ; Type: Library
 INITLONG 10,xxxName ; LibName
 INITBYTE 14,6 ; Flags: SUMUSED & CHANGED
 INITWORD 20,Version ; Versionsnummer
 INITWORD 22,Revision ; Revisionsnummer
 INITLONG 24,xxxLibID ; LibIDString
 dc.l 0  ; End
```

Die Revisionsnummer ist eine Überarbeitungsnummer, die dann erhöht wird, wenn an der Library kleine Änderungen vorgenommen wurden, die keine Erhöhung der Versionsnummer rechtfertigen. Andere Daten werden nicht initialisiert, weil dies das System beim Erstellen der Library macht. Beim dritten Eintrag handelt es sich um einen Zeiger auf eine »InitLib«-Routine. Sie wird beim ersten Öffnen der Library aufgerufen. Das Betriebssystem übergibt uns im Register d0 die Basisadresse unserer Library und in a0 die Adresse der »Segmentliste«. Beides sollten wir auf jeden Fall merken. Normalerweise wird die Adresse der Segmentliste in der Basis der Library gespeichert und die Basisadresse merkten wir uns in der Variablen _xxxBase (xxx steht für den Namen Ihrer Library). Der Pflichtteil von InitLib ist also folgender:

```
InitLib movem.l a1/a5,-(sp)
 move.l d0,a5
 lea.l _xxxBase(pc),a1
 move.l d0,(a1)
 move.l a0,34(a5)
 move.l a5,d0
EndInit movem.l (sp)+,a1/a5
 rts
```

Wenn für die Befehle und Funktionen der eigenen Library noch andere Libraries, wie in unserem Beispiel die »graphics.library«, benötigt werden, können diese hier geöffnet werden. Es empfiehlt sich, die Basisadressen der Libraries in der eigenen Library-Basis zu speichern.

Die Sprungtabelle ist entweder eine Langwort- oder eine Worttabelle. Wählen Sie die Langworttabelle, stehen in ihr Zeiger auf die Befehle und Funktionen der Library. Möchten Sie aber eine Worttabelle benutzen, steht als erstes:

```
dc.w -1
```

Die folgenden Einträge bestehen dann aus Adreßdistanzen zwischen Beginn der Sprungtabelle und den jeweiligen Routinen. Beide Tabellen werden durch

```
dc.l -1
```

beendet. Die Wort-Tabelle für unsere Beispiel-Library sieht dann so aus:

```
FunctionTable
 dc.w -1
 dc.w Open - FunctionTable
 dc.w Close - FunctionTable
 dc.w ExpugneLib - FunctionTable
 dc.w ExtFuncLib - FunctionTable
 dc.w DrawTriangle - FunctionTable
 dc.w GetAPen - FunctionTable
```

```
dc.w GetBPen - FunctionTable
dc.l -1
```

Und die Langworttabelle so:

```
FunctionTable       dc.l        Open
dc.l Close
dc.l ExpugneLib
dc.l ExtFuncLib
dc.l DrawTriangle
dc.l GetAPen
dc.l GetBPen
dc.l -1
```

Die ersten vier Einträge zeigen auf Routinen die normalerweise nur intern gebraucht werden:

❒ Open: Diese Routine wird immer dann vom System aufgerufen, wenn ein Task die Library mit »OpenLibrary()« öffnen möchte. Wenn wir in der Open-Routine landen, erhöhen wir den Inhalt von Adresse Basis+32 (der Inhalt dieser Adresse gibt an, wieviele Tasks die Library im Moment benutzen) und löschen in den Library-Flags Bit 3. Die benötigte Basisadresse liefert uns das System in Register a6. Diese müssen wir vor dem Rücksprung aus der Routine ins Register d0 übertragen. Demnach sieht die Open-Routine mindestens so aus:

```
Open addq.w #1,32(a6)
 bclr.b #3,14(a6)
 move.l a6,d0
 rts
```

❒ Close: Die Routine stellt das Gegenteil der Open Routine dar, d.h. – wie sollte es auch anders sein –, daß sie immer dann angesprungen wird, wenn ein Task die Library mittels »CloseLibrary()« schließt. Hier muß allerdings noch überprüft werden, ob hiermit der letzte Task die Library geschloßen hat. Ist das der Fall, wird die »ExpugneLib()«-Routine aufgerufen (s.u.). Also sieht die Close-Routine z.B. so aus:

```
Close subq.w #1,32(a6)
 btst.b #3,14(a6)
 beq.s EndClose
 bsr.s Expugne
EndClose rts
```

❒ ExpugneLib: Die Routine kann entweder bei Speicherknappheit von anderen Tasks oder von der Close-Routine aufgerufen werden. Bei ExpugneLib und auch bei Close liefert uns das Betriebssystem in a6 die Basisadresse unserer Library. Als erstes wird hier überprüft, ob kein Task die Library mehr benutzt. Ist das nicht der Fall, wird Bit 3 der Library-Flags gesetzt und dann erfolgt der Rücksprung mit einer 0 als Returncode in d0.

Wenn allerdings der letzte Task die Bibliothek geschloßen hat, wird eine Routine angesprungen, die in unserem Beispiel »DiscardLib()« heißt. Sie räumt alles auf, d.h., daß eventuell bei InitLib geöffnete Libraries wieder geschlossen werden und unsere Library aus der Systemliste der geöffneten Libraries entfernt wird. Das geschieht dadurch, daß wir mit der Basisadresse unserer Library im Register a1 »Remove()« aus der »exec.-library« aufrufen. Nun holen wir uns die Adresse der Segmentlist aus der Basis zurück und merken sie uns z.B. im Register d7 oder auf dem Stack. Ist dies geschehen, bringen wir die Basisadresse unserer Library ins Register a0, schreiben die Größe der Sprungtabelle in Register d0 und ziehen d0 von a0 ab. Dann addieren wir die Größe der Basis auf d0 und rufen »FreeMem()« auf (exec.-library) auf. Zu guter Letzt bringen wir die Adresse der Segmentliste ins Register d0 und springen zurück. Eine Expugne-Routine könnte also so aussehen:

```
ExpugneLib movem.l d7/a1/a5/a6,-(sp)
 move.l a6,a5
 tst.w 32(a5)
 beq.s DiscardLib
 bset #3,14(a5)
 clr.l d0
ExpEnd  movem.l (sp)+,d7/a1/a5/a6
 rts
DiscardLib move.l _SysBase,a6
 move.l a5,a1
 jsr _LVORemove(a6)
 move.l 34(a5),d7
 clr.l d0
 move.l a5,a1
 move.w 16(a5),d0
 sub.l d0,a1
 add.w 18(a5),d0
 jsr _LVOFreeMem(a6)
 move.l d7,d0
 bra.s ExpEnd
```

❒ ExtFuncLib: Diese Routine ist für künftige Erweiterungen reserviert und wird noch nicht genutzt. Deshalb schreiben wir sicherheitshalber in das Register d0 eine 0 und springen zurück.

Bleibt noch die Größe der Basis zu erklären. Sie errechnet sich folgendermaßen:

```
34 (für den Systemteil der Struktur) + Zahl
```

Zahl steht für die Größe der eigenen Daten in Byte. Es gibt nämlich die Möglichkeit, die Basis beliebig zu vergrößern, um darin Daten wie Basisadressen anderer Libraries unterzubringen.

Assembler-Routinen von BASIC aus nutzen

Der Systemteil der Library-Basis setzt sich wie in der Tabelle rechts oben zusammen:

Damit wäre dann der Rumpf der Library fertig. Nun können Sie nach Herzenslust (so lange der Speicher reicht) Befehle und Funktionen programmieren. Beginnen Sie jeden mit einer Sprungmarke und tragen Sie dann dieselbe in der Sprungtabelle direkt hinter die internen Routinen ein.

Das Listing »xxx.library.asm« zeigt, wie das Ganze in Assembler aussieht. Das zugehörige BASIC-Programm »xxxTest.bas« demonstriert den Einsatz von Routinen aus der eigenen Bibliothek von BASIC aus. Allgemein zur Programmierung ist noch zu sagen:

– PC-relative Programmierung ist angesagt !! (die Branch-Befehle sind von Natur aus PC-relativ)

Offset (Dez.)	Größe	Bedeutung
0	struct	Hier ist eine Node-Struktur eingebunden
14	byte	Die Libraryflags
15	byte	Ein Füllbyte
16	word	Größe der Sprungtabelle
18	word	Größe der Basis
20	word	Versionsnummer
22	word	Revisionsnummer
24	aptr	Zeiger auf Identifikationstring
28	long	Checksumme der Library
32	word	Anzahl der Tasks, welche die Library nutzen
34	wählbar	für eigene Daten

Die Libraryflags:

Wert	Bedeutung
1	Gerade wird die Checksumme berechnet (brauchen wir nicht)
2	Checksumme muß neu berechnet werden
4	Checksumme soll vom Betriebssystem überprüft werden
8	Library soll entfernt werden; wird aber noch benutzt

Die Node-Struktur:

Offset (Dez.)	Größe	Bedeutung
0	aptr	(hier) Zeiger vorhergehende Library
4	aptr	(hier) Zeiger nachfolgende Library
8	byte	Da dies eine Lib_Node ist, wird 9 eingetragen
9	byte	Priorität: normalerweise 0
10	aptr	(hier) Zeiger Library- Namen

Der Systemteil einer Library im Detail (siehe auch Listing folgende Seite)

– Returncodes von Funktionen müssen im Register d0 geliefert werden. Bei Befehlen sollte man auf alle Fälle d0 löschen.

– Bei jeder Library-Routine müssen die benutzten Register auf den Stack gerettet werden. Wenn das unterbleibt, werden sich die Tasks, die Ihre Library gebrauchen, höchstwahrscheinlich mit einem Display-Alert bedanken. Davon ausgenommen sind natürlich die sog. Schmierregister also d0 und a0. Eigentlich gehöhren d1 und a1 auch dazu, worüber man sich allerdings auch streiten kann. Außerdem ausgenommen sind die Register, in denen wir von den aufrufenden Tasks Werte entgegennehmen.

Wenn wir nun die Library nach unseren Wünschen fertiggestaltet haben, müssen wir eine »xxx_lib.fd« schreiben (Für unser Bsp. zu finden auf der PD-Diskette zum Heft; siehe Seite 114). Diese wird benötigt, um mit dem Programm »MakeInclude« das Include-File für die Assembler-Programmierer und mit dem bekannten ConvertFD das ».bmap«-File für die BASIC-Programmierer zu generieren. Eine »_lib.fd« zu schreiben, ist eigentlich nich so schwer, denn es gibt nur sieben Kommandos. In jeder Zeile darf nur eines stehen:.

Erstes Kommando (»*«): Durch den Stern wird eine Bemerkung eingeleitet. Nach einem Stern wird also der Rest der Zeile ignoriert.

Zweites Kommando ("): Wenn die Zeile weder mit »*« noch mit »##« beginnt, wird

die Zeile so interpretiert: Als erstes steht der Name der Funktion oder des Befehls unter dem Sie von BASIC aus aufgerufen werden. Der Name geht bis dahin, wo die erste Klammer auf (»(«) erscheint. In dieser Klammer stehen die Namen der zu übergebenden Werte, die aber nur zu Dokumentationszwecken gut sind. Die Namen werden durch Kommas getrennt angegeben. In der nächsten Klammer werden die Register (wieder durch Kommas getrennt) angegeben, in die die Werte übergeben werden sollen. Wichtig: Die Namen der Routinen müssen in der lib.fd-Datei in gleicher Reihenfolge angegeben werden, wie sie in der Sprungtabelle der Library stehen!

Libraries machen selbst BASIC schnell

Alle folgenden Kommandos werden durch »##« am Anfang der Zeile gekennzeichnet.

Drittes Kommando (»base«): Nach diesem Kommando steht ein Leerzeichen und dann der Name der Basis unserer Library.

Viertes Kommando (»bias«): Der Wert hinter diesem Kommando gibt an, wieviele Routinen in der Sprungtabelle für interne Nutzung bestimmt sind.

```
Wert = (Anzahl der internen Routinen + 1)*6
```

Gewöhnlich sind es vier Routinen, wie oben beschrieben. Das heißt: Wert = (4 + 1)*6 also Wert = 30

Fünftes Kommando (»end«): Dieses Kommando sagt dem Programm, daß es seine Arbeit beenden kann.

Sechstes Kommando (»public«): Alle nachfolgenden Routinen können von mehreren Tasks gleichzeitig genutzt werden.

Siebtes und letztes Kommando (»private«): Alle nachfolgenden Routinen können immer nur von einen Task zur Zeit genutzt werden. Sie können dann aber selbstverständlich von mehreren Tasks hintereinander angesprungen werden. Aber dieses Kommando wird nur in ganz speziellen Fällen gebraucht.

Die _lib.fd für unsere Beispiel-Library sieht demnach folgendermaßen aus:

```
* die lib.fd wird benötigt, um die .bmap
  zu erzeugen
* Und diese braucht AmigaBASIC zum
  Ansprechen der Routinen.
##base _xxxBase
##bias 30
##public
DrawTriangle(x1,y1,x2,y2,x3,y3,RastPort)
            (d0,d1,d2,d3,d4,d5,a1)
GetAPen(RastPort)(a1)
GetBPen(RastPort)(a1)
##end
```

Wenn Sie Ihre eigene Library erstellen, setzen Sie bitte überall, wo wir in diesem Kurs »xxx« verwendet haben, den Namen Ihrer Library ein. So das war's dann auch schon! Wie sagt doch der Mathematiker; »Es ist alles ganz einfach, wenn man halt weiß, wie's geht !!« *ub*

```
 1: RANDOMIZE TIMER
 2: LIBRARY "df0:Library/xxx.library"
 3: DECLARE FUNCTION GetAPen&(RastPort) LIBRA
    RY
 4: DECLARE FUNCTION GetBPen&(RastPort) LIBRA
    RY
 5: REM *** DrawTriangle ist keine Funktion s
    ondern ein Befehl
 6: RastPort& = WINDOW(8)
 7: COLOR 2,3:PrintColor
 8: DrawTriangle& 110,50,10,100,210,100,RastP
    ort&
 9: COLOR 1,0:LOCATE 2,1:PrintColor
10: DrawTriangle& CINT(RND*610),CINT(RND*186)
    ,CINT(RND*610),CINT(RND*186),CINT(RND*610
    ),CINT(RND*186),RastPort&
11: LIBRARY CLOSE
12: LOCATE 3,1
13: END
14:
15: SUB PrintColor STATIC
16: SHARED APen%,BPen%,RastPort&
17:    APen% = GetAPen&(RastPort&)
18:    BPen% = GetBPen&(RastPort&)
19:    PRINT "Vordergrundfarbe ist"APen%"und
   Hintergrundfarbe ist"BPen%
20: END SUB
21:
22: © 1993 M&T
```

»xxxTest.bas«: Das Listing zeigt, wie wir unsere Library von BASIC nutzen

```
  1: Version  equ 1
  2: Revision equ 1
  3:   include ":include/graphics_lib.i"
  4:   include ":include/exec_lib.i"
  5: INITBYTE macro
  6:   dc.b $e0
  7:   dc.b 0
  8:   dc.w \1
  9:   dc.b \2
 10:   dc.b 0
 11:   endm
 12: INITWORD macro
 13:   dc.b $d0
 14:   dc.b 0
 15:   dc.w \1
 16:   dc.w \2
 17:   endm
 18: INITLONG macro
 19:   dc.b $c0
 20:   dc.b 0
 21:   dc.w \1
 22:   dc.l \2
 23:   endm
 24:   move.l #20,d0
 25:   rts
 26: ResidentStruct dc.w 19196  ; o. 'illegal
     ' o. 'dc.w $4AFC'
 27:   dc.l ResidentStruct
 28:   dc.l End
 29:   dc.b 128
 30:   dc.b 1
 31:   dc.b 9
 32:   dc.b 0
 33:   dc.l xxxName
 34:   dc.l xxxLibID
 35:   dc.l Start
 36: Start  dc.l 42  ; Größe der Basis = 34+L
     änge der Daten
 37:   dc.l FunctionTable
 38:   dc.l DataTable
 39:   dc.l InitLib
 40: FunctionTable dc.l Open
 41:   dc.l Close
 42:   dc.l ExpugneLib
 43:   dc.l ExtFuncLib
 44:   dc.l DrawTriangle
 45:   dc.l GetAPen
 46:   dc.l GetBPen
 47:   dc.l -1
 48: DataTable INITBYTE 8,9  ; Type:Library
 49:   INITLONG 10,xxxName ; LibName
 50:   INITBYTE 14,6  ; Libraryflags
 51:   INITWORD 20,Version
 52:   INITWORD 22,Revision
 53:   INITLONG 24,xxxLibID ; LibIDString
 54:   dc.l 0  ; End
 55: InitLib  movem.l a1/a5,-(sp)
 56:   move.l d0,a5
 57:   lea.l _xxxBase(pc),a1
 58:   move.l d0,(a1)
 59:   move.l a0,34(a5)
 60:   lea.l gfxname(pc),a1
 61:   clr.l d0
 62:   jsr _LVOOpenLibrary(a6)
 63:   tst.l d0
 64:   beq.s InitFailed
 65:   move.l d0,38(a5)
 66:   move.l a5,d0
 67: EndInit  movem.l (sp)+,a1/a5
 68:   rts
 69: InitFailed clr.l d0
 70:   bra.s EndInit
 71: Open  addq.w #1,32(a6)
 72:   bclr.b #3,14(a6)
 73:   move.l a6,d0
 74:   rts
 75: Close  subq.w #1,32(a6)
 76:   btst.b #3,14(a6)
 77:   beq.s EndClose
 78:   bsr.s ExpugneLib
 79: EndClose rts
 80: ExpugneLib movem.l d7/a1/a5/a6,-(sp)
 81:   move.l a6,a5
 82:   tst.w 32(a5)
 83:   beq.s DiscardLib
 84:   bset #3,14(a5)
 85:   clr.l d0
 86: ExpEnd  movem.l (sp)+,d7/a1/a5/a6
 87:   rts
 88: DiscardLib move.l _SysBase,a6
 89:   move.l 38(a5),a1
 90:   jsr _LVOCloseLibrary(a6)
 91:   move.l a5,a1
 92:   jsr _LVORemove(a6)
 93:   move.l 34(a5),d7
 94:   clr.l d0
 95:   move.l a5,a1
 96:   move.w 16(a5),d0
 97:   sub.l d0,a1
 98:   add.w 18(a5),d0
 99:   jsr _LVOFreeMem(a6)
100:   move.l d7,d0
101:   bra.s ExpEnd
102: ExtFuncLib clr.l d0
103:   rts
104: DrawTriangle movem.l a5/a6,-(sp) ; gebra
     ute Register retten
105:   movem.l d0/d1,-(sp) ; d0 und d1 werden
      noch gebraucht
106:   move.l a1,a5  ; RastPort aufheben
107:   move.l _xxxBase(pc),a6 ; Basis der gra
     phics.library
108:   move.l 38(a6),a6 ; holen
109:   jsr _LVOMove(a6)
110:   move.l d2,d0
111:   move.l d3,d1
112:   move.l a5,a1  ; RastPort zurück
113:   jsr _LVODraw(a6)
114:   move.l d4,d0
115:   move.l d5,d1
116:   move.l a5,a1  ; und noch einmal
117:   jsr _LVODraw(a6)
118:   movem.l (sp)+,d0/d1 ; d0 und d1 zurück
119:   move.l a5,a1  ; und wieder RastPort zu
     rück
120:   jsr _LVODraw(a6)
121:   clr.l d0  ; kein Returncode
122:   movem.l (sp)+,a5/a6 ; Register zurück
123:   rts
124: GetAPen  clr.l d0
125:   move.b 25(a1),d0
126:   rts
127: GetBPen  clr.l d0
128:   move.b 26(a1),d0
129:   rts
130: xxxName  dc.b 'xxx.library',0
131:   even
132: xxxLibID dc.b 'Frei wählbarer Identifika
     tionsstring',13,10,0
133:   even
134: gfxname  dc.b 'graphics.library',0
135:   even
136: _xxxBase ds.l 1
137: End
138:   END
139:
140: © 1993 M&T
```

»xxx.library.asm«: So programmiert man eine Bibliothek in Assembler

Programmierkniffe und -hilfen

Tips, Tricks und tolle Tools

Die Rubrik Tips & Tricks ist die meistgelesene und -begehrteste im AMIGA-Magazin. Jeden Monat finden Sie dort die besten Tips und Kniffe, die Amiga-Besitzer an die AMIGA-Redaktion schicken. Gerade für Programmierer gibt´s dort immer ein paar Rosinen, die das Programmieren auf dem Amiga erleichtern oder über – scheinbar unüberbrückbare – Hürden helfen. Auf den folgenden Seiten nun die neuesten – bisher unveröffentlichten – Tips... speziell für Programmierer.

Mit den Tips ist es manchmal wie mit Nüssen und Austern (und Wildschweinen[1]): die kleinen sind die besten. Deshalb hier nun eine Reihe von Tips, kurzen Programmen und Anregungen, die Sie in Ihren Programmen einsetzen sollten. Alles buntgemischt in C, Assembler usw., so daß für jeden etwas dabei sein sollte.

Falls Sie selbst ein paar Tips auf Lager haben, mit denen Sie sich am nächsten Sonderheft »Faszination Programmieren« beteiligen möchten, schicken Sie Ihren Beitrag mit allen Listings etc. auf Diskette an:

AMIGA-Redaktion,
Markt & Technik Verlags AG,
Hans-Pinsel-Str. 2,
Stichwort: Faszination Programmieren

Viel Erfolg und natürlich viel Spaß mit den Tips dieser Ausgabe.

[1]: Quelle: Asterix als Legonär

Mit Makros immer geradeaus

Das Zeichnen von Linien über die »graphics.library« setzt normalerweise die Verwendung von zwei Funktionen voraus, da zunächst der Grafic-Cursor positioniert werden muß und dann erst eine Linie zu einem weiteren Punkt gezeichnet werden kann. Es handelt sich um die Funktionen »Move()« und »Draw()«. Beide können durch eine Hilfsfunktion zusammengefaßt werden, um Linien mit einem Aufruf zu zeichnen, was auch z.B. in AmigaBASIC möglich ist:

```
void DrawLine(rp,x1,y1,x2,y2)
struct RastPort rp;
SHORT x1,y1,x2,y2;
{ Move(rp,x1,y1); Draw(rp,x2,y2); }
```

Neben dem Anfangs- und Endpunkt, also (x1|y1) und (x2,y2), muß der Rastport übergeben werden, in den die Linie gezeichnet werden soll. An den Rastport gelangt man über die Window-Struktur, die für jedes Intuition-Fenster existiert und die man beim Öffnen mit »OpenWindow()« erhält:

```
RastPort=IrgendEinWindow->RPort;
```

Christoph Brühann

Boolean-Variable mit Makro toggeln

Variable vom Typ »boolean« (kurz BOOL) können immer nur den Wahrheitswert TRUE (1) oder FALSE (0) enthalten. Manchmal muß der Inhalt einer solchen Variablen getoggelt werden, d.h. abhängig vom Zustand wird von TRUE nach FALSE oder umgekehrt von FALSE nach TRUE gewechselt. Für diesen Schritt ist eine IF-ELSE-Anweisung naheliegend, die den Zustand abfragt und die Variable dann je nach Ergebnis mit dem anderen Wahrheitswert beschreibt.

Mit folgendem Markro geht es durch Verwendung des Bedingungsoperators »?« jedoch wesentlich eleganter:

```
#define ToggleBool(a) a = a ? FALSE : TRUE
```

Der Bedingungssoperator hat allgemein den Aufbau:

```
op1 ? op2 : op3
```

Das Ergebnis des Ausdrucks ist dabei abhängig vom Operator op1. Ist er ungleich Null, erhält das Ergebnis des gesamten Ausdrucks den Wert von op2, ansonsten von op3. Überträgt man das auf ToggleBool(), wird die Arbeitsweise des Makros deutlich.

Christof Brühann/irw

Speicher reservieren mit System

Speicher reservieren unter Kickstart V2.0: Unter Kickstart V1.3 ist es üblich, Speicher mit der Funktion »AllocMem()« zu reservieren und mit »FreeMem()« wieder freizugeben. Mit Kickstart V2.0 sind die Funktionen »AllocVec()« sowie »FreeVec()« hinzugekommen, die im wesentlichen das gleiche bewirken, jedoch noch einen entscheidenden Vorteil besitzen: Beim Reservieren über »AllocVec()« hält der Amiga die Größe des Speicherblocks fest, so daß die Freigabe unter Weglassung der Größe geschehen kann. Dadurch wird nicht nur der Programmieraufwand geringer, sondern es werden auch Fehler beim Freigeben durch falsche Speichergrößen ausgeschlossen.

Die Parameter bei »AllocVec()« haben sich gegenüber »AllocMem()« nicht geändert, die Umstellung ist deshalb einfach. Man muß beim Freigeben lediglich darauf achten, daß »FreeVec() nur der Zeiger auf den freizugebenen Speicherblock zu übergeben ist.

Christof Brühann/irw

»SizeDIR« zeigt echte Größe

Das folgende Listing »SizeDIR.c« ermöglicht Ihnen die Bestimmung der Größe eines Verzeichnisses, wobei sämtliche Unterverzeichnisse mit beliebiger Tiefe berücksichtigt werden.

Name der Quelldatei: SizeDir.c
Aufruf mit DICE: dcc SizeDir.c -o SizeDir
Name des ausführbaren Programms: SizeDir

```
 1: /* SizeDir.c - von Christof Brühann
 2:    Aufruf mit DICE: dcc SizeDir.c -o SizeDir
 3:                     SizeDir <Verzeichnis>  */
 4:
 5: #include <libraries/dos.h>
 6: #include <exec/memory.h>
 7: ULONG size=0,numdirs=1,numfiles=0;
 8:
 9: BOOL SizeDir(path)
10:     /* Bestimmt Größe eines Verzeichnisses */
11: UBYTE *path;    /* mit Unterverzeichnissen */
12: {UBYTE nxtpath[50];
13:     /* Pfad für nächstes Unterverzeichnis */
14:  struct FileLock *lock;
15:  struct FileInfoBlock *fib;
16:  if (fib=(struct FileInfoBlock *)AllocMem(siz
    eof(struct FileInfoBlock),MEMF_PUBLIC))
17:   /* Speicher für FileInfoBlock besorgen */
18:  {if (lock=(struct FileLock *)Lock(path,ACCES
    S_READ))
19: /* Lock auf zu untersuchendes Verzeichnis */
20:   {if (Examine(lock,fib)) /* mehr Info */
21:    {while (ExNext(lock,fib))
22:            /* nächster Verzeichniseintrag */
23:     {numfiles++; /* Zähler für Dateien */
24:      size+=fib->fib_Size; /* Größe erhöhen */
25:      if (fib->fib_DirEntryType>0)
26:                  /* Unterverzeichnis ? */
27:      {numfiles--; /* dann neuen Pfad er- */
28:       numdirs++;  /* stellen und SizeDir() */
29:       strcpy(nxtpath,path); /* aufrufen  */
30:       strcat(nxtpath,fib->fib_FileName);
31:       strcat(nxtpath,"/");
32:       SizeDir(nxtpath);
33:      }
34:     }
35:    }
36:    UnLock(lock);   /* Lock wieder freigeben*/
37:   }
38:   else return(FALSE);
39:       /* Lock konnte nicht besorgt werden */
40:   FreeMem(fib,sizeof(struct FileInfoBlock));
41:             /* Speicher für FileInfo frei */
```

```
42:  }
43:  return(TRUE);
44: }
45: void main(argc,argv)
46: SHORT argc;
47: UBYTE **argv;
48: {
49:  UBYTE path[50];
50:  path[0]=0;
51:  if (argc>1)    /* Parameter vorhanden ? */
52:  {
53:   strcpy(path,argv[1]);
54:   printf("\nBestimme Größe vom Verzeichnis '%
    s'.\n\n",argv[1]);
55:   if (path[strlen(argv[1])-1]!=':') strcat(pa
    th,"/");
56:  }
57:  else printf("\nBestimme Größe vom aktuellen
    Verzeichnis.\n\n");
58:  if (SizeDir(path))   /* Hauptfunktion aufruf
    en */
59:  {     /* und Ergebnisse ausgeben */
60:   printf("%d Bytes wurden mit %d Dateien in %
    d Verzeichnis",size,numfiles,
61:           numdirs);
62:   if (numdirs>1) printf("sen");
63:   printf(" erreicht.\n\n");
64:  }
65:  else printf("Zugriff auf Verzeichnis '%s' ni
    cht möglich.\n\n",argv[1]);
66: }
67:                          © 1993 M&T
```

»SizeDir«: Das Programm ermittelt die Gesamtgröße eines Verzeichnisses

Sie können dem Programm ein Verzeichnis, dessen Größe bestimmt werden soll, als Parameter übergeben. Es kann aber auch das bei Programmaufruf aktuelle Verzeichnis herangezogen werden, wenn kein Parameter angegeben wird.

»SizeDir« errechnet die Größe jeder einzelnen Datei und addiert sie zur Gesamtgröße. »SizeDir« arbeitet mit einem rekursiven Algorithmus. Er ermöglicht es, in eine beliebig tiefe Verzeichnisstruktur zu steigen, was im DICE-Listing »SizeDir.c« mit der Funktion »SizeDir()« erreicht wird. Beim Untersuchen der einzelnen Verzeichniseinträge mit den entsprechenden DOS-Funktionen prüfen wir für jeden Eintrag, ob er für ein Verzeichnis steht. Bei Verzeichnissen wird dann die Funktion »SizeDir()« einfach nochmal aufgerufen, und zwar mit dem neuen Unterverzeichnis als Parameter. In diesem Aufruf gehen wir genauso vor wie beim ersten Mal, wobei auch hier bei Unterverzeichnissen gleichermaßen verfahren wird.

Christof Brühann/irw

Es muß nicht immer PRT: sein

Bei den meisten Ein- und Ausgaben stellt das Betriebssystem Devices zur Verfügung, um ein multitaskinggerechtes Arbeiten zu ermöglichen. Für den Drucker gibt es beispielsweise das sog. Printer-Device, welches die Ausgabe sowohl von Grafiken als auch Texten erlaubt. Doch was das Drucken von Texten betrifft, ist das Printer-Device programmtechnisch gar nicht nötig, wodurch das relativ aufwendige Erstellen von Message-Port und Device-Block entfällt.

Die Lösung liefert die »DOS.library«, die alle Funktionen zur Ein- und Ausgabe enthält. Der Trick besteht aus dem richtigen Öffnen des benötigten File-Handles mit der Funktion »Open()«. Als Name wird hier nämlich einfach der Name PRT: angegeben, als Modus MODE_OLDFILE:

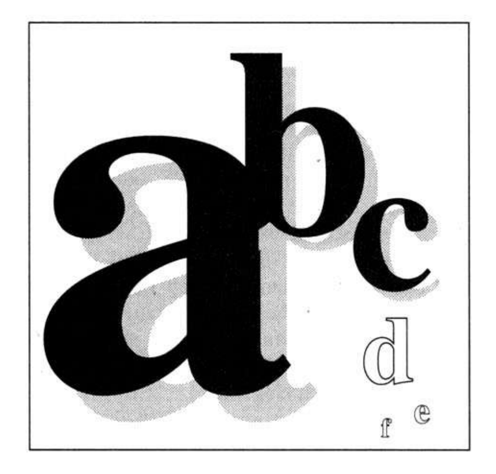
```
file_prt=(struct FileHandle *)
          Open("prt:",MODE_OLDFILE);
```

Mit dem nun vorhandenen File-Handle können Sie alle gewünschten Ausgaben zum Drucker schicken, was mit DOS wie gewöhnlich mit der Funktion »Write()« geschieht und für einen String so aussieht:

```
Write(file_prt,String,strlen(String);
```

Übrigens halten wir beim Drucken über DOS die Programmierkonventionen ein, obwohl das Printer-Device – scheinbar – nicht benutzt wird. Tatsächlich jedoch setzt der Amiga es ein, weil »PRT:« die DOS-Kennzeichnung für das Printer-Device ist. Versuchen Sie aber auch ruhig, »PAR:« statt »PRT:« einzusetzen.

Christof Brühann/irw



Fonts in jeder beliebigen Größe

Die »diskfont.library« bietet ab V.36 die Möglichkeit, Pixelfonts auch zu vergößeren oder zu verkleinern. Das aber verhindert ein Programm mit gesetztem FPF_DESIGNED. Des weiteren sind viele Programme so ausgelegt, daß ihr Fontmenü nur die Auswahl der vom System gemeldeten Größen zuläßt (kein AFF_SCALED). Als Abhilfe erzeugt man den Font in der gewünschten Größe.

Geben Sie hierzu in ein Shell-Fenster

```
SetFont <Name> <Größe> SCALE [PROP]
```

ein. Die Option PROP ist nur für Proportional-Fonts (z.B. ruby) erforderlich.

Der Befehl SetFont ist nur für das Shell-Fenster, in dem er aufgerufen wird, gültig. Die Standardtexteinstellung im Font-Editior wird für dieses Fenster außer Kraft gesetzt, das Fenster gelöscht und die Eingabeaufforderung erscheint in der neuen Schriftart.

Anschließend lädt man den skalierten Font in den Font-Editor »FEd« (befindet sich auf der Extras1.3-Disk) mit dem Aufruf

```
Extras1.3D:Tools/FEd
```

und speichert ihn. Damit steht diese Größe auch als Pixel-Font im Verzeichnis FONTS: zur Verfügung.

Arno Eigenwillig/irw

Ergänzung durch die Redaktion: Mit der Batch-Datei »fontsize.bat« läßt sich der Ablauf automatisieren.

```
.key font,size
setfont <font> <size> SCALE PROP
echo "Das ist:" <font> <size>
if exists "Extras1.3D:Tools/FEd" then
 Extras1.3D:Tools/FEd
 setfont topaz 8
else
 setfont topaz 8
 echo "*n Extras1.3D:Tools/FEd nicht gefunden *n"
endif     ;           © 1993 M&T
```

»fontsize.bat«: Befehlsdatei, um Fonts in beliebiger Größe zu erzeugen

Referenzparameter in C

Beim Aufrufen von Funktionen mit Parametern in einer Hochsprache werden i.a. die übergebenen Variablen nicht von der aufgerufenen Funktion geändert. Anders bei sogenannten Referenzparameter; in der Sprache Pascal durch das Schlüsselwort »VAR« gekennzeichnet. Hier wird für die angegebene Variable in der aufgerufenen Funktion kein neuer Speicherplatz reserviert, es gibt somit keine lokale Variable innerhalb der Funktion. Stattdessen ist die übergebene Variable in der Funktion dieselbe wie im aufrufenden Teil, d.h. bei Änderungen des Variableninhalts ist nach Beendigung der Funktion die Variable im aufrufenden Teil ebenso geändert. Die in Pascal verbreitete Benutzung von Referenzparametern läßt sich auch auf die Sprache C übertragen. Zwar existiert hier kein eigenes Schlüsselwort, doch sind Referenzparameter in C möglich. Das Prinzip besteht darin, die Adresse der Variablen zu übergeben, da so auch von der aufgerufenen Funktion auf die Variable zugegriffen werden kann. Dazu braucht vor dem Parameter nur der Adressoperator stehen:

```
main() {SHORT i; IrgendEineFunktion(&i);}
```

Nun muß im Kopf der aufgerufenden Funktion natürlich noch gekennzeichnet werden, daß nicht der Inhalt einer SHORT-Variable, sondern die Adresse auf so eine Variable übergeben wurde:

```
void IrgendEineFunktion(j) SHORT *j; {...}
```

Die Variable, die nicht wie im aufrufenden Teil heißen muß, enthält nun die Adresse auf unsere SHORT-Variable. Beim Zugriff auf die Variable muß das beachtet werden, und zwar muß beim Beschreiben der Inhaltsoperator eingesetzt werden:

```
*j = 42;
```

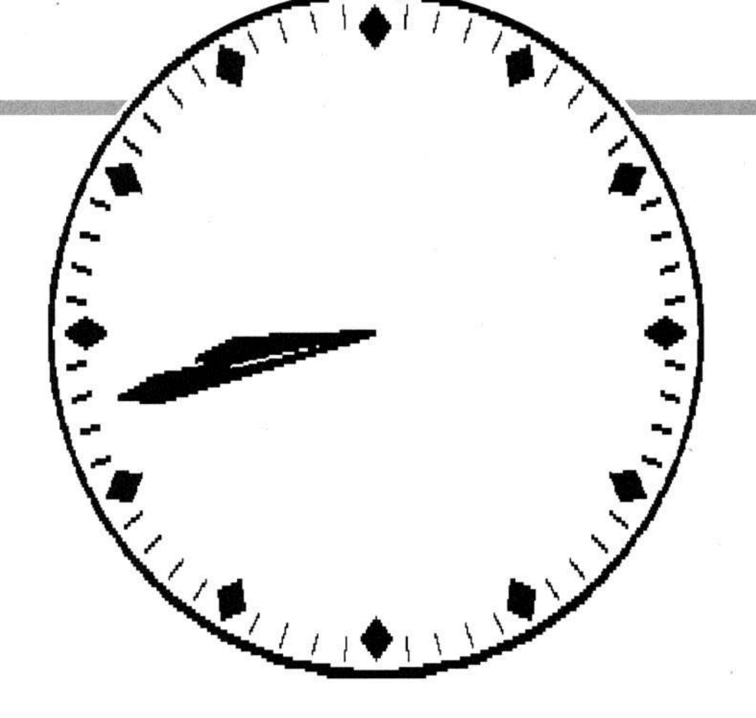
Schließlich soll ja die Adresse auf die Variable nicht geändert werden, sondern lediglich ihr Inhalt. Dieser ist wiederum identisch mit dem Inhalt der Variable i aus dem main-Teil, womit das Problem der Benutzung von Referenzparameter in C gelöst ist.

Christof Brühann

Warum »DateTime«? Weil einige Programme die Workbench beim Start schließen wollen und das mit den anderen Uhren nicht schaffen, weil ein Window geöffnet ist.

Daß manchmal die Workbench geschlossen werden muß, merkt man z.B. am Programm Populous II. Wenn bei diesem Programm die Workbench nicht geschlossen werden kann, treten Fehlfunktionen auf. Bevor man Populous II aufruft, muß man daher erst immer das Fenster mit der Uhr schließen und nach dem Beendigen des Spiels die Uhr wieder laden.

Das Programm wurde unter OS1.3 und OS2.0 getestet. *Andreas Baum/irw*



Immer die richtige Zeit mit DateTime

Es gibt nicht nur Uhrensammler, es gibt auch Sammler von Uhrenprogrammen. Hier ist eines für die Sammlung:

»Date-Time« zeigt das Datum und die Uhrzeit nicht in einem Fenster, sondern in der Workbench-Titelzeile.

Quelldatei: Date-Time.c
Aufruf mit DICE: siehe Listing
Ausführbares Programm: siehe Listing

```
 1: /* Date-Time.c
 2:  * 06.10.92 by Andreas Baum
 3:  * Das Programm erzeugt eine Datums- und Uhr-
 4:  * anzeige in der Titelzeile des Workbench-
 5:  * bildschirms. Das hat den Vorteil, daß
 6:  * Programme, die den Workbenchbildschirm
 7:  * schließen wollen, auf kein offenes Fenster
 8:  * treffen, wie es bei anderen Uhren
 9:  *
10:  * Compiler DICE Version 2.06.38
11:  * Compileraufruf:
12:  * dcc -oRAM:Date-Time Date-Time.c
13:  * Programmaufruf:
14:  * 1. Workbench   :  Doppelklick auf Icon
15:  * ( mit IconEdit erzeugen [ TOOLICON ] )
16:  * 2. CLI / Shell :  Run >NIL: Date-Time
17:  * Ein erneuter Aufruf beendet das Programm*/
18:
19:
20: #include <intuition/intuition.h>
21: #include <intuition/intuitionbase.h>
22: #include <exec/execbase.h>
23: #include <exec/memory.h>
24: #include <workbench/startup.h>
25: #include <time.h>
26:
27: extern struct  IntuitionBase *IntuitionBase;
28: extern struct  ExecBase      *SysBase;
29: extern struct  WBStartup     *_WBMsg;
30:
31: struct Screen   *scr,   *scr1;
32: struct Window   *win;
33: struct Task     *task;
34: struct tm       *zeit;
35:
36: struct NewWindow nwin=
37:  {0,11,250,11,0,1,NULL,
38:   ACTIVATE|RMBTRAP,NULL,NULL,NULL,NULL,NULL,
39:   0,0,0,0,WBENCHSCREEN
40:  };
41:
42: UBYTE buff[80], buff1[25];
43: UBYTE taskname[10] = "D-T.0001";
44: UBYTE winnam1[27]="DATE-TIME BY ANDREAS BAUM"
    ;
45: UBYTE winnam2[28]="THANKS FOR USING DATE-TIME
    ";
46: _main()
47:  {UBYTE x=0, y=0, ver;
48:    if ( FindTask ( "D-T.0001" ) == 0 )
49:     { task = ( struct Task * ) FindTask ( NUL
    L );
50:       task->tc_Node.ln_Name = taskname;
51:       nwin.Title = winnam1;
52:       win = ( struct Window * ) OpenWindow (
    &nwin );
53:       scr = win->WScreen;
54:       Delay ( 80 );
55:       CloseWindow ( win );
56:       if ( SysBase->LibNode.lib_Version < 37 )
57:          ver = 1;
58:        else
59:          ver = 2;
60:          FOREVER
61:          { time_t just;
62:            time ( &just );
63:            zeit = localtime ( &just );
64:            for ( y = 0 ; y < 15 ; y++ )
65:             { if ( ( strlen ( scr->Title ) < 6
    5 || zeit->tm_min != x ) && ( strncmp ( scr->
    Title, "Amiga Workbench  ", 17 ) == 0 || strn
    cmp ( scr->Title, "Workbench release.", 18 )
    == 0 ) )
66:                { x = zeit->tm_min;
67:                  strftime ( buff1, 25, "%d %b
    %y  %H:%M", zeit );
68:                  if ( ver == 1 )
69:                    sprintf ( buff, "Workbench
    release.     %d free memory      %s", AvailM
    em ( MEMF_PUBLIC ), buff1);
70:                  else
71:                    sprintf ( buff, "Amiga Work
    bench  %d graphics mem  %d other mem  %s", Av
    ailMem ( MEMF_CHIP ), AvailMem ( MEMF_FAST ),
     buff1);
72:                    scr->Title = buff;
73:                    ShowTitle ( scr, TRUE );
74:                    }
75:                  Delay ( 10 );
76:                }
77:              scr1 = IntuitionBase->ActiveScreen;
78:                if ( scr1 != scr )
79:                  {
80:                   if ( strncmp ( scr1->Title, "
    Workbench Screen", 16 ) == 0 || strncmp ( scr
    1->Title, "Amiga Workbench  " ,17 ) == 0 || s
    trncmp ( scr->Title, "Workbench release.", 18
     ) == 0 )
81:                    scr = scr1;
82:                  }
83:              }
84:        }
85:     else
86:        {
87:          Forbid();
88:          RemTask ( FindTask ( "D-T.0001" ) );
89:          Permit();
90:
91:          nwin.Title = winnam2;
92:          win = ( struct Window * ) OpenWindow
    ( &nwin );
93:          Delay ( 80 );
94:          CloseWindow ( win );
95:
96:          exit ( 0 );
97:        }
98:     _waitwbmsg();
99:   }     ;                © 1993 M&T
```

Date-Time.c: Hiermit zeigt der Amiga die Uhrzeit in der Titelzeile

Speicher grafisch sichtbar machen

Das Programm »MemoryViewer.c« macht den Speicher grafisch sichtbar. Mehr noch: Durch Bewegungen eines am Gameport 2 angeschlossenen Joysticks können Sie den Workbench-Screen verschieben und der angrenzende Speicherbereich wird sichtbar. So lassen sich sogar im Speicher befindliche Bilder betrachen. Ein Druck auf den Feuerknopf stellt den Workbench-Screen wieder her und beendet das Programm. ⇨

Quelldatei : MemoryViewer.c
Aufruf mit DICE:
dcc MemoryViewer.c -o MemoryViewer
Ausführbares Programms: MemoryViewer

```
 1: /* MemoryViewer.c
 2: Mit DICE: dcc MemoryViewer.c -o MemoryViewer
 3:    Start:     MemoryViewer
 4: */
 5: #include <intuition/intuitionbase.h>
 6: #include <graphics/gfxbase.h>
 7: #include <graphics/view.h>
 8: #include <graphics/rastport.h>
 9: #include <hardware/custom.h>
10: #include <hardware/cia.h>
11: #define custom (*((struct Custom *)0xdff000))
12: #define ciaa (*((struct CIA *)0xbfe001))
13: #define MOD(a,b) (a-a/b*b)
14: struct IntuitionBase *IntuitionBase;
15: struct GfxBase *GfxBase;
16: struct ViewPort *ViewPort;
17: PLANEPTR OldPlanes[4];
18:
19: void main()
20: {
21:  struct BitMap *BitMap;
22:  USHORT i,width,joydat,button;
23:  GfxBase=(struct GfxBase *)OpenLibrary("graph
    ics.library",0);
24:  /* Libraries öffnen */
25:  IntuitionBase=(struct IntuitionBase *)OpenLi
    brary("intuition.library",0);
26:  ViewPort=(struct ViewPort *)ViewPortAddress(
    IntuitionBase->ActiveWindow);
27:  /* BitMap des Screens ermitteln */
28:  BitMap=ViewPort->RasInfo->BitMap;
29:  for (i=0;i<4;i++) OldPlanes[i]=ViewPort->Ras
    Info->BitMap->Planes[i];
30:  /* Adressen der Bitplanes sichern */
31:  if (MOD(ViewPort->DWidth,16)) width=(ViewPor
    t->DWidth/16+1)*4;
32:   else width=ViewPort->DWidth/4;
33:   /* Breite der Planes in Bytes bestimmen */
34:  Forbid();/*nicht mehr in Planes schreiben*/
35:  do
36:  {joydat=custom.joy1dat;
37:  /* Datenregister für Joystick 1 auslesen */
```

```
38:   if (joydat&2) /* Bewegung nach rechts ? */
39:   {ViewPort->DxOffset++;
40:    /* Viewport um einen Pixel verschieben */
41:    if (ViewPort->DxOffset==16)
42:    /* beim nächsten Datenwort Adresse der */
43:    {  /* Planes ändern */
44:     for (i=0;i<4;i++) BitMap->Planes[i]-=2;
45:     ViewPort->DxOffset=0;
46:    }
47:   }
48:   if (joydat&512)  /* Bewegung nach links ? *
    /
49:   {ViewPort->DxOffset--;
50:    if (ViewPort->DxOffset==-16)
51:    {ViewPort->DxOffset=0;
52:     for (i=0;i<4;i++) BitMap->Planes[i]+=2;
53:    }
54:   }
55:   if (((joydat&1)==1)^((custom.joy1dat&2)==2)
    )
56:                       /* Bewegung nach unten */
57:    for (i=0;i<4;i++) ViewPort->RasInfo->BitMa
    p->Planes[i]-=width;
58:   if (((joydat&256)==256)^((joydat&512)==512)
    )
59:                        /* Bewegung nach oben */
60:    for (i=0;i<4;i++) ViewPort->RasInfo->BitMa
    p->Planes[i]+=width;
61:   ScrollVPort(ViewPort);
62:                /* Viewport neu darstellen */
63:   button=ciaa.ciapra;
64:  } while (button&CIAF_GAMEPORT1);
65:                            /* Feuerknopf ? */
66:  for (i=0;i<4;i++) ViewPort->RasInfo->BitMap-
    >Planes[i]=OldPlanes[i];
67:  /* Ausgangszustand wiederherstellen */
68:  ViewPort->DxOffset=0;
69:  ViewPort->DyOffset=0;
70:  ScrollVPort(ViewPort);
71:  Permit();
72: }
73:                     © 1993 M&T
```

Listing: Diff.c: Das C-Programm leitet einen mathematischen Ausdruck ab

Die Basis des Programms wird durch die Funktion »ScrollVPort« der »graphics.library« gebildet. Die Funktion stellt den Viewport neu dar, wobei Veränderungen in der Viewport-Struktur berücksichtigt werden.

Ausgehend vom Viewport, der die Workbench darstellt, werden je nach Joystick-Bewegung der Eintrag »DxOffset« oder die Bitplane-Adressen geändert und es entsteht eine Verschiebung des Screens. Bei Bewegungen nach links oder rechts wird zunächst nur der Eintrag »DxOffset« geändert. Erreicht »DxOffset« den Umfang eines Datenworts (16 Punkte), setzt der Amiga den Offset auf 0 zurück und ändert dafür die Adressen der Bitplanes um die entsprechende Anzahl von Bytes.

Bewegt man den Joystick nach oben oder unten, werden einfach die Bitplane-Adressen um die Breite des Viewports in Bytes erhöht beziehungsweise erniedrigt.

Die Bitplane-Adressen sind in der Bitmap-Struktur eingetragen, welche über

```
Viewport->RasInfo->BitMap
```

besorgt wird.

Weil das Programm die Adressen der Bitplanes ändert, darf nichts in den Rastport geschrieben werden, so lange ein unbekannter Speicherbereich angezeigt wird. Andernfalls überschriebe der Amiga unbekannte Speicherstellen. Deshalb wird das System – ausnahmsweise mal – mit »Forbid()« angehalten, bis wir am Ende des Programms den Ausgangszustand wiederherstellen.

Christof Brühann/irw

Sortieren nach der Zeit

Einige Amiga-Anwender werden sicherlich ein Programm vermissen, daß ein Verzeichnis nach Zeit sortiert ausgibt. Der bereits vorhandene CLI-Befehl LIST erlaubt zwar eine sortierte Ausgabe, allerdings besteht nur die Möglichkeit, die Dateien nach ihren Namen zu sortieren, was durch das Programm »SortByDate.c« geändert werden soll.

Quelldatei : SortByDate.c
Aufruf mit DICE: siehe Listing
Ausführbares Programms: siehe Listing

```
  1: /*SortByDate.c - von Christof Brühann
  2:   Betriebssystem: Kickstart V2.04
  3:   Kompilieren mit DICE:
  4:   dcc SortByDate.c -o SortByDate */
  5:
  6: #include <dos/dos.h>
  7: #include <exec/memory.h>
  8: #include <time.h>
  9: #include <dos/var.h>
 10: #define FIBSIZE sizeof(struct FileInfoBlock)
 11: char *mon[]={"Jan","Feb","Mär","Apr","Mai","
    Jun","Jul","Aug","Sep","Okt","Nov","Dez"};
 12: UBYTE dir[255]={0};
 13: struct FileInfoBlock *fib,*fibptr[250];
 14:
 15: void Usage() /* Info über Aufruf ausgeben */
 16: {
 17:  printf("\nSortByDate [<verzeichnis>] [<anza
    hl>]\n\n");
 18:  printf("<verzeichnis> - auszugebenes Verzei
    chnis (Voreinstellung: aktuelles ");
 19:  puts("Verzeichnis)");
 20:  printf("<anzahl>      - Anzahl auszugebener
     Einträge (Voreinstellung: alle)");
 21:  printf("\n\n");
 22: }
 23: void main(argc,argv)
 24: SHORT argc;
 25: UBYTE **argv;
 26: {
 27:  LONG num2list=250,numall=0,i,j;
 28:  time_t secs;
 29:  struct tm *tm;
 30:  struct Lock *lock;
 31:  APTR mem;
 32:  if (strcmp(argv[1],"?")==0 || argc>3) Usage
    ();
 33:  /* ggf. Programminfo ausgeben */
 34:  else
 35:  {
 36:   if (argc==2)  /* Parameter untersuchen*/
 37:    if (sscanf(argv[1],"%d",&num2list)==-1)
 38:                /* und Variablen für */
 39:    {           /* Verzeichnis und Anzahl*/
 40:     num2list=250;  /* beschreiben */
 41:     strcpy(dir,argv[1]);
 42:    }
 43:
 44:   if (argc==3) /*zwei Parameter vorhanden?*/
 45:   {
 46:    if (sscanf(argv[1],"%d",&num2list)==-1)
 47:    {
 48:     strcpy(dir,argv[1]);
 49:     num2list=0;
 50:     sscanf(argv[2],"%d",&num2list);
 51:    }
 52:    else strcpy(dir,argv[2]);
 53:   }
 54:   if (num2list<1) Usage();
 55:   mem=(APTR)AllocMem(num2list*FIBSIZE,MEMF_P
    UBLIC);
 56:   /* Speicher für Verzeichniseinträge */
 57:   fib=(struct FileInfoBlock *)AllocMem(FIBSI
    ZE,MEMF_PUBLIC);
 58:   if (mem&&fib)   /* genug Speicher? */
 59:   {
 60:    for (i=0;i<num2list;i++) fibptr[i]=mem+i*
    FIBSIZE;
 61: /*Zeiger->Verzeichniseinträge beschreiben*/
 62:    if (lock=(APTR)Lock(dir,ACCESS_READ))
 63:     /* Lock auf Verzeichnis holen */
 64:     {
 65:      if (Examine(lock,fib))
 66:      /* nähere Infos über Verzeichnis */
 67:       if (fib->fib_DirEntryType>0)
 68:  /*Parameter für Verzeichnis eine Datei? */
 69:       {
 70:        while (ExNext(lock,fib))
 71:         /* nächster Eintrag */
 72:        {
 73:         numall++;
 74:          /* Zähler für gefundene Einträge */
 75:         if (numall==1) memcpy(fibptr[0],fib,F
    IBSIZE);
 76:         else
 77:         {
 78:  j=num2list;
 79:  if (numall<num2list) j=numall;
 80:  for (i=0;i<j;i++)
 81: /* Eintrag in vorhandene Liste einordnen:*/
 82:  {
 83:   if (CompareDates(&fibptr[i]->fib_Date,&fib
    ->fib_Date)>=0)
 84:       /* bzgl. Revisionsdatum vergleichen */
 85:   {
 86:    memcpy(fibptr[i+1],fibptr[i],(j-i-1)*FIBS
    IZE);
 87:       /* einordnen */
 88:    memcpy(fibptr[i],fib,FIBSIZE);
 89:    i=-1;
 90:    break;
 91:   }
 92:  }
 93:  if ((i!=-1) && (numall<=num2list))
 94:   memcpy(fibptr[numall-1],fib,FIBSIZE);
 95:         }
 96:        }
 97:        j=num2list;
 98:     /* Ausgabe der sortierten Liste: */
 99:        if (numall<num2list) j=numall;
100:        for (i=j-1;i>=0;i--)
101:        {
102:         printf("%-25s     ",fibptr[i]->fib_Fi
    leName);
103:         if (fibptr[i]->fib_DirEntryType>0) pr
    intf("%7s","Dir");
104:   else printf("%7d",fibptr[i]->fib_Size);
105:         secs=fibptr[i]->fib_Date.ds_Days*(144
    0*60)+fibptr[i]->fib_Date.ds_Minute*
106:         60+fibptr[i]->fib_Date.ds_Tick/50;
107: /* Umwandlung von DateStamp nach Sekunden*/
108:         tm=localtime(&secs);
109:    /* tm-Struktur ausfüllen lassen */
110:         printf(" %02d.%3s.%02d  %02d:%02d:%0
    2d",tm->tm_mday,mon[tm->tm_mon],
111:         tm->tm_year,tm->tm_hour,tm->tm_min,tm
    ->tm_sec);
112:         printf("\n");
113:        }
114:        printf("%d von %d Einträgen ausgegeben
    .\n",j,numall);
115:       }
116:       else printf("'%s' ist kein Verzeichnis
    !\n",fib->fib_FileName);
117:      UnLock(lock);
118:     }
119:     else puts("Falsches Verzeichnis !");
120:    }
121:    if (mem) FreeMem(mem,num2list*FIBSIZE);
122:    if (fib) FreeMem(fib,FIBSIZE);
123:  }
124: }     ;                        © 1993 M&T
```

»SortByDate.c«: Sortierte Ausgabe eines Verzeichnisses per C-Programm

Der Aufruf dieses in C verfaßte Programms lautet:

```
SortByDate [<verzeichnis>] [<anzahl>]
```

Beide Parameter sind optional. Der erste gibt das Verzeichnis an, das ausgegeben werden soll, wobei bei fehlender Angabe das aktuelle Verzeichnis verangezogen wird. Soll das gesamte Verzeichnis ausgegeben werden, kann der zweite Parameter weggelassen werden. Möchte man jedoch nur eine bestimmte Anzahl der neusten Einträge ausgeben, kann das mit diesem Parameter getan werden. So ist es z.B. möglich, durch Angabe von 1 nur die zuletzt bearbeitete Datei auszugeben.



Beim Listen wird neben dem Namen auch die Größe des Eintrags in Bytes sowie der Zeitpunkt der letzten Bearbeitung ausgegeben. Verzeichnisse können durch die Kennung »Dir« anstelle der Größe von Dateien unterschieden werden. »SortByDate« richtet sich zur Ermittlung des letzen Bearbeitungszeitpunkts nach der DateStamp-Struktur, die im FileInfoBlock eingetragen ist. Diese wird bei jedem Schreibzugriff vom Betriebssystem automatisch modifiziert. Da man an den FileInfoBlock sowieso immer dann gelangt, wenn ein Verzeichnis über Examine() und ExamineNext() ausgegeben wird, ist das Beschaffen dieses Datenblockes für die einzelnen Einträge kein Problem. Jeder gefundene Eintrag wird in eine Liste geschrieben. Sollte ein Eintrag wegen seines Bearbeitungsdatums nicht ans Ende der Liste angefügt werden können, muß dieser einsortiert werden. Dies wird erreicht, indem alle Einträge hinter der entsprechenden Postion nach hinten verschoben werden, so daß der neue Eintrag anschließend direkt an die richtige Position geschrieben werden kann. Mit diesem Verfahren sind bereits alle Einträge sortiert, wenn das Verzeichnis vollständig gelesen ist. *Christof Brühann*

Der Mauszeiger bleibt kleben

Im Handbuch zur System-Software 2.0 wird im Kapitel 2-6 beschrieben, wie man den Amiga auch ohne Maus betreiben kann. Doch es gibt auch undokumentierte Funktionen: Halten Sie die linke Amiga-Taste und danach die linke Maus-Taste gedrückt (die Reihenfolge ist wichtig!). Die Position der Maus ist bedeutungslos. Nun klebt der Mauszeiger auf dem Screen fest und kann mit der Maus vertikal wenn möglich auch horizontal verschoben werden. *Felix Farago/irw*

Mit LIST und ohne Tücken

Vor dem Start eines Programms vom CLI aus ist es oft sinnvoll, das Verzeichnis, in dem sich das Programm befindet, mit DIR oder LIST anzuschauen. Das kann verschiedene Gründe haben: Zum einen wird so sichergestellt, daß das Programm auch wirklich im entsprechenden Verzeichnis zu finden ist. Zum anderen kennt man oft nicht den vollständigen Name des Programms, der zum Aufruf schließlich komplett eingegeben werden muß. Besonders bei neuen und deshalb noch nicht vertrauten Programmen, was meistens bei neuer Software aus dem PD-Bereich der Fall ist, trifft das immer wieder zu.

In letzterem Fall weiß man oft auch nicht, welche Datei das Hauptprogramm ist und zum Start aufgerufen werden muß. Beim Auflisten des Verzeichnisses ist es dann störend, daß auch reine Daten- und Info-Dateien angezeigt werden. Sie bewirken nur, daß man den Überblick verliert.

Name der Quelldatei : ListExecutables.c
Übersetzen mit DICE: dcc ListExecutables.c -o
Ausführbaren Programm: ListExecutables

```
 1: /* ListExecutables.c - listet alle ausführbar
    en Programme eines Verzeichnisses
 2: Aufrufe mit DICE:
 3:   dcc ListExecutables.c -o ListExecutables
 4:       ListExecutables <directory>
 5: */
 6:
 7: #include <dos/dos.h>
 8:
 9: struct Lock *lock;
10: struct FileInfoBlock fib;
11: struct FileHandle *file;
12:
13: void main(argc,argv)
14: SHORT argc;
15: UBYTE **argv;
16: {ULONG type;
17:  UBYTE path[255],i;
18:  if (lock=(APTR)Lock(argv[1],ACCESS_READ))
19:              /* Lock auf Verzeichnis holen */
20:  {if (Examine(lock,&fib))
21:                   /* Info über Verzeichnis */
22:    if (fib.fib_DirEntryType>0)
23:              /* war Parameter Verzeichnis ?*/
24:    {i=strlen(argv[1]);
25:     /* wenn nötig, an Pfad '/' anhängen */
26:     strcpy(path,argv[1]);
27:     if (path[i-1]!=':' && path[i-1]!='/' && s
   trlen(argv[1])) path[i++]='/';
28:     while (ExNext(lock,&fib))
29:     /* nächste Datei des Verzeichnisses */
30:     {path[i]=0;
31:      strcat(path,fib.fib_FileName);
32:                /* Pfad mit Datei erstellen */
33:      if (file=(APTR)Open(path,MODE_OLDFILE))
34:                         /* und Datei öffnen */
35:      {if (Read(file,&type,4)==4)
36:                /* Dateityp prüfen und Name */
37:        if (type==0x3f3) /* ggf. ausgeben */
38:  printf ("%s\n",fib.fib_FileName);
39:       Close(file);
40:      }
41:     }
42:    }
43:    else printf("'%s' ist kein Verzeichnis !\n
   ",fib.fib_FileName);
44:   UnLock(lock);
45:  }
46:  else puts("Falsches Verzeichnis !");
47:  /* Parameter beschreibt kein Verzeichnis*/
48: }
49:
```


»ListExecutables.c«: Das C-Programm listet alle ausführbaren Programme

Für solche Fälle ist ein Utility nützlich, das nur Programme auflistet, die tatsächlich gestartet werden können. Das DICE-Programm »ListExecutables.c« übernimmt dieses Funktion. Es listet alle Dateien vom Typ »executable« (ausführbar) auf.

Beim Aufruf von ListExecutables ohne Angabe eines Verzeichnisses wird das aktuelle Verzeichnis oder das als Parameter angegebene Verzeichnis untersucht.

Bei der Auflistung geht »ListExecutables« prinzipiell vor wie der normale LIST-Befehl, da auch hier die entsprechenden DOS-Funktionen »Lock()«, »Examine()« sowie »ExtNext()« verwendet werden. Allerdings werden nur die Dateien ausgegeben, die ausführbar sind. Sie sind daran erkennbar, daß sie mit einem Hunk vom Typ »Load-File« beginnen, welches wiederum durch ein »0x3f3« als erstes Langwort identifiziert wird. *Christof Brühann/irw*

Parameterauswertung in Assembler

Viele zusätzliche Informationen können bei CLI-Programmen durch die Verwendung von Parametern übergeben werden. Möchte ein Programmierer diese Möglichkeit auch für seine Programme nutzen, muß die richtige Vorgehensweise zur Auswertung der Argumente bekannt sein. Da in C über die Funktion main() mit den Parametern argc sowie argv bereits der Compiler das fertige Parameter-Array zur Verfügung stellt, fällt der Zugriff auf die einzelnen Argumente hier nicht schwer.

Ein wenig anders ist es in Assembler, da hier zunächst nur der gesamte Parameter-

String sowie seine Länge in den Registern A0 und D0 vorhanden ist. Doch stellt das neue Betriebssystem V2.0 zahlreiche neue Funktionen zur Verfügung, welche die ganze Sache vereinfachen. Ziel soll es sein, daß jeder einzelne Parameter als String in einem Parameter-Array vorliegt. Auf diese Weise kann die Auswertung der Parameter später dann durch gezielte Vergleiche mit den einzelnen Parametern vorgenommen werden. Am einfachsten wird die Aufteilung des Parameter-Strings durch die neue Dos-Funktion ReadItem() erreicht. Ausgehend von einem beliebigen Argument-String liest diese Funktion ein einziges Argument heraus und schreibt dieses als String in einen Puffer. Dabei wird der Argument-String nicht als einfacher String angegeben, sondern als eine CSource-Struktur, die folgendermaßen aussieht:

```
struct CSource {
UBYTE *CS_Buffer;  ; Zeiger auf Puffer
LONG   CS_Length;  ; Länge des Puffers
LONG   CS_CurChr; ; akt.Zeichenposition
 };
```

In den Einträgen CS_Buffer und CS_Length wird die Adresse des Parameter-Strings sowie seine Länge eingetragen. Dabei können die Register A0 und D0 am Anfang des Programms direkt übernommen werden. Der Vorteil bei Verwendung der CSource-Struktur gegenüber eines einfachen Strings liegt im Eintrag CS_CurChr begründet, der immer mit der aktuellen Zeichenposition im Puffer beschrieben ist und anfangs den Wert 0 bekommt. Die Funktion »ReadItem()« berücksichtigt nämlich beim Ablaufen diesen Eintrag. D.h. in der Praxis, daß das nächste aufzulösende Argument immer an Position CS_CurChr beginnt, wobei »ReadItem()« CS_CurChr automatisch erhöht. Zum Auflösen des gesamten Parameterstrings braucht deshalb »ReadItem()« nur solange aufgerufen werden, bis das Ende erreicht ist. Lediglich der Zielpuffer muß bei jedem Durchgang erhöht werden. Als Ergebnis erhält man ein Array mit allen Argumenten. »ReadArguments.s« ist das zugehörige Demoprogramm.

```
1: OpenLibrary: equ -552
2: ReadItem: equ -810
3: VPrintf: equ -954
4: ITEMSIZE: equ 40
5:
6: move.l a0,ArgStrg ; Zeiger auf Argument-String
7: move.l d0,ArgStrgLen  ; und Länge retten
8: move.l 4,a6
9: lea DosName,a1
10: move.l #37,d0 ; mindestens OS 2.0 erforderlich
11: jsr OpenLibrary(a6)  ; Dos-Library öffnen
12: tst.l d0   ; V37 nicht vorhanden ?
13: beq quit
14: move.l d0,DosBase ; Library-Basis retten
15: move.l ArgStrg,a0 ; Parameter f. ReadArguments()
16: move.l ArgStrgLen,d0
17: jsr ReadArguments  ; ReadArguments() aufrufen
18: lea param,a0 ; Zahl Argumente in Param.-Array
19: move.l NumItems,(a0)  ; schreiben
20:        ; (wird für VPrintf() gebraucht)
21: move.l #fmsg1,d1
22: move.l #param,d2
23: move.l DosBase,a6
24: jsr VPrintf(a6) ; Zahl Argumente ausgeben
25: lea ItemArray,a3  ; Adresse des Arrays der Argumente
26: sub.l d3,d3
27: out_args:   ; Argumente ausgeben:
28: cmp.l NumItems,d3 ; alle Argumente ausgegeben ?
29: beq quit
30: lea param,a0 ; für VPrintf() wieder Parameter-Array
31: move.l d3,(a0)   ; beschreiben (mit Argumentnummer
32: move.l a3,4(a0)   ; und Adresse des Arguments )
33: move.l #fmsg2,d1
34: move.l #param,d2
35: jsr VPrintf(a6)   ; Ausgabe des Arguments
36: add.l #1,d3   ; Argumentnummer erhöhen
37: add.l #ITEMSIZE,a3 ; Zeiger->nächstes Argument
38: bra out_args   ; weiter mit nächstem Argument
39: quit:
40: sub.l d0,d0
41: rts
42:
43: ; ReadArguments() - unterteilt den Argument-String in die verschiedene
44: ;      Argumente und schreibt sie in das Array "ItemArray"
45: ; Parameter: a0: Argument-String
46: ;       d0: Länge des Argument-Strings
47: ReadArguments:
48: movem.l d2-d3/a3/a6,-(sp)
49: move.l DosBase,a6
50: lea CSource,a1  ; Argument-String und seine Länge in CSource-Struktur
51: move.l a0,(a1)  ; eintragen, die der Dos-Funktion ReadItem()
52: move.l d0,4(a1)  ; übergeben wird
53: move.l #0,8(a1)
54: lea ItemArray,a3 ; Adresse des Argument-Arrays
55: clr.l NumItems ; Zähler für Anzahl der Argumente
56: read_item:
57: move.l ArgStrgLen,d0
58: sub.l #1,d0
59: lea CSource,a0
60: cmp.l 8(a0),d0  ; schon fertig ?
61: beq quit_ReadArguments
62: move.l a3,d1  ; Ziel für nächstes Argument
63: move.l #ITEMSIZE,d2
64: move.l #CSource,d3
65: jsr ReadItem(a6) ; nächstes Argument bestimmen mit ReadItem()
66: add.l #ITEMSIZE,a3 ; Adresse nächstes Argument
67: add.l #1,NUMITEMS ; Argumentenzähler erhöhen
68: bra read_item  ; nächste Runde
69: quit_ReadArguments:
70: movem.l (sp)+,d2-d3/a3/a6
71: rts
72:
73: ItemArray: blk.b ITEMSIZE*50
74: even
75: NumItems: dc.l 0
76: ArgStrgLen: dc.l 0
77: ArgStrg: dc.l 0
78: CSource: blk.b 12
79: DosName: dc.b "dos.library",0
80: even
81: DosBase: dc.l 0
82: fmsg1:  dc.b 13,10,"Anzahl der Argumente: %ld",13,10,10,0
83: even
84: fmsg2:  dc.b "Argument %ld: %s",13,10,0
85: even
86: param:  dc.l 0,0     ;
```


»ReadArguments.s«: Beispiel für Parameterauswertung in Assembler

Die Hauptfunktion wird durch »ReadArguments()« gebildet. Ihr ist der Parameterstring in A0 sowie die Länge in D0 zu übergeben. Sie liefert als Ergebnis das Array ab Label »ItemArray« mit den Argumenten.

Die maximale Größe eines Arguments wird mit ITEMSIZE festgegt. Das Argument Nummer n (n=0, 1, ..., NumItems-1) befindet sich demnach an der Adresse

`ItemArray+n*ITEMSIZE`

Die Anzahl der vorhandenen Argumente wird in der Variable »NumItems« gespeichert. Das Hauptprogramm gibt nach dem Aufruf von »ReadArguments()« zunächst die Anzahl der Argumente aus, dann die Argumente selbst. Bei der Ausgabe wird dabei die ebenfalls neue DOS-Funktion »VPrintf()« verwendet, welche eine formatierte Ausgabe einfach macht. Probieren Sie das Programm »ReadArguments« ruhig mal aus, indem Sie es vom CLI aus starten und verschiedene Parameter übergeben. Übrigens hat die Verwendung der OS-Funktion »ReadItem()« große Vorteile gegenüber der Programmierung von Hand, da »ReadItem()« alle vorhandenen Konventionen beachtet und somit die Programme standardisiert. Z.B. wird durch ein Semikolon jeder Parameterstring beendet. Dadurch lassen sich Kommentare verwenden, was auch bei allen anderen CLI-Befehlen möglich ist. Außerdem wird das Gleichheitszeichen als Leerzeichen interpretiert. Zwar läßt sich unter Berücksichtigung des Rückgabewerts von ReadItem() dies unterscheiden, doch kann man z.B. auch beim CLI-Befehl LIST statt

`list s: sort`

auch

`list s:=sort`

schreiben! Zum Schluß sei noch erwähnt, daß mit »ReadItem()« bei Verwendung von Anführungsstrichen auch Argumente erlaubt sind, die sich aus mehreren Wörtern zusammensetzen, was ebenfalls standardisiert ist.

Christof Brühann/irw

Zeig alles, was Du hast

Mit dem Hilfsprogramm »all.c« kann man die Wirkung von AmigaDOS-Befehlen auf Unterverzeichnisse ausdehnen, d.h. ein gewünschter Befehl wird nicht nur auf das aktuelle oder angegebene Verzeichnis angewandt, sondern auch auf sämtliche Unterverzeichnisse. Dazu durchläuft es rekursiv den angegebenen Verzeichnisbaum und führt die als Paramater übergebenen Befehle aus.

Der Aufruf erfolgt folgendermaßen:

`all [-A] [-E] Verzeichnis Befehl1 [Befehl2 Befehl3 ...]`

Der Parameter »Verzeichnis« bestimmt dabei den zu durchlaufenden Verzeichnisbaum. Für das aktuelle Verzeichnis kann man wie unter AmigaDOS üblich "" angeben.

Mit »Befehl1 [Befehl2 Befehl3 ...]« ist die Befehlsfolge gemeint, die in den Verzeichnissen des Verzeichnisbaums ausgeführt werden soll. Natürlich muß zumindest ein Befehl angegeben werden.

Es ist zu beachten, daß unter AmigaDOS keine Anführungszeichen übergeben werden können. »all« bietet jedoch die Möglichkeit, anstelle eines Anführungszeichens ein Hochkomma (<Alt ä>) zu verwenden. Damit das

Hochkomma selbst nicht wegfällt, stehen zwei aufeinanderfolgende Hochkomma für ein richtiges Hochkomma. Außerdem ist zu beachten, daß ein Befehl mit all seinen Parametern als ein einzelner Parameter an »all« übergeben werden muß. Ein – wenn auch wenig sinnvolles – Beispiel soll diese beiden Punkte verdeutlichen:

```
all df0: "echo 'Hallo'"
```

gibt für jedes Verzeichnis auf df0: das Wort »Hallo« aus.

Die optionalen Parameter »[-A] [-E]« legen fest, in welcher Reihenfolge die Verzeichnisse abgearbeitet werden. Wählen Sie beispielsweise -E, wird mit dem Ende des Verzeichnisbaums begonnen; bei -A mit dem Anfang. -A ist voreingestellt, muß also nicht explizit angegeben werden. Gibt man

```
all -A df0: cd
```

und danach

```
all -E df0: cd
```

ein, wird der Unterschied schnell klar.

Der Programmablauf kann natürlich durch <Ctrl C> abgebrochen werden. Bei einem Fehler oder bei einem Abbruch gibt »all« den Fehlercode 10 an seine Umgebung zurück. »all« ohne einen Parameter gibt eine gekürzte Version der Anleitung aus.

Was kann man nun konkret mit dem Programm anfangen? Hierzu eine nützliche Batch-Datei mit Namen »FIND«:

```
.key Verzeichnis/a,file/a,opt1,opt2,opt3,opt4
resident c:echo add
. 'cd' und 'list' hat wohl jeder resident
failat 15
. sonst bleibt echo bei Fehler resident
all <Verzeichnis> "echo '*E[1m' noline" cd
   "echo '*E[0m' noline" "list pat <file>
   <opt1> <opt2> <opt3> <opt4> nohead"
resident c:echo remove
```

FIND durchsucht einen Verzeichnisbaum nach einer bestimmten Datei. Im Gegensatz zu anderen Programmen dieser Art geschieht die Ausgabe über den AmigaDOS LIST-Befehl. Dem Benutzer stehen somit alle Möglichkeiten des LIST-Kommandos offen. Man kann mit den Optionen von LIST leicht die Ausgabe beeinflussen und sämtliche Amiga DOS-Wildcards verwenden. Aufruf:

```
FIND Verzeichnis Dateimuster [Optionen]
```

Ein Beispiel:

```
FIND df0: d#?
```

Der Aufruf gibt sämtliche Dateien auf df0, welche mit »d« oder »D« beginnen sowie die dazugehörigen Verzeichnisnamen aus.

```
FIND ">prt: df0:" d#?
```

gibt dieselben Dateien auf den Drucker aus. Wer sich intensiver mit »all« beschäftigt, wird sicher noch viele andere nützliche Verwendungsmöglichkeiten finden. Ausgesprochen nützlich ist dabei die LFORMAT-Funktion des LIST-Befehls. Bsp.:

```
all df0: "list >T:qwe #? LFORMAT='protect
        %s -d'" "execute T:qwe"
delete T:qwe
```

Die beiden Zeilen schützen alle Dateien auf DF0: davor gelöscht zu werden.

that's »all« *Daniel Görtz/ub*

```
  1: /* -geschrieben unter Kickstart V1.3
  2:    -läuft auch unter Kickstart V2.0
  3:    -übersetzt mit Aztec C-Compiler V5.0
  4:     Aufruf: cc all -pa -wd -wp -wr -wu
  5:             ln all c.lib
  6:    -läßt sich auch mit DICE übersetzen */
  7: #include <string.h>
  8: #include <stdlib.h>
  9: #include <functions.h>
 10: #include <libraries/dos.h>
 11: #include <libraries/dosextens.h>
 12: #define PUT(s)  Write((BPTR) OutputFile, s
     , (long) strlen(s))
 13: #define BREAK   ((SetSignal(0L, 0L) & 4096
     L) == 4096L)
 14: #define HI_ON "\033[33m"
 15: #define HI_OFF "\033[0m"
 16: enum Fehler { Kein_Fehler, Speicher_Fehler
     , Disk_Fehler, Befehl_Fehler, Falscher_Auf
     ruf, Abbruch };
 17: enum wann { Anfang, Ende };
 18: void Aufruf(char *FName, struct FileHandle
      *OutputFile)
 19: { PUT("Aufruf:  "HI_ON); PUT(FName);
 20:   PUT(HI_OFF"  [-A] [-E] Verzeichnis Befeh
     l1 [Befehl2 Befehl3 ...]\n");
 21:   PUT("\t "); PUT(FName);
 22:   PUT(" durchläuft rekursiv einen Verzeich
     nisbaum und führt\n");
 23:   PUT("\t dabei die angegebene Befehlsfolg
     e aus.\n");
 24:   PUT("\t Verzeichnis    Name des Verzeich
     nisses, welches\n");
 25:   PUT("\t                den Verzeichnisba
     um festlegt.\n");
 26:   PUT("\t                \"\" steht fuer d
     as aktuelle Verzeichnis.\n");
 27:   PUT("\t Befehl1        DOS-Befehle, die
     in den Verzeichnissen des\n");
 28:   PUT("\t [Befehl2 ...]  Verzeichnisbaumes
      ausgeführt werden sollen.\n");
 29:   PUT("\t                Mindestens ein Be
     fehl muß angegeben werden.\n");
 30:   PUT("\t                Da unter AMIGA-DO
     S keine Anführungszeichen\n");
 31:   PUT("\t                übergeben werden
     können, muß man stattdessen\n");
 32:   PUT("\t                ein Hochkomma (Al
     t+Ä) übergeben.\n");
 33:   PUT("\t                \'\' steht für da
     s Hochkomma selber.\n");
 34:   PUT("\t [-A] [-E]      Über diese option
     alen Parameter wird festgelegt in\n");
 35:   PUT("\t                welcher Reihenfol
     ge die Verzeichnisse abgearbeitet\n");
 36:   PUT("\t                werden. Wird -E g
     ewählt, so wird mit dem Ende des\n");
 37:   PUT("\t                Verzeichnisbaumes
      begonnen; bei -A mit dem Anfang.\n");
 38:   PUT("\t                -A ist voreingest
     ellt, muß also nicht explizit\n");
 39:   PUT("\t                angegeben werden.
     \n\n");
 40:   PUT("\t                Tritt ein Fehler
     auf oder wird durch CTR-C abge-\n");
 41:   PUT("\t                brochen, liefert
     "); PUT(FName);
 42:                          PUT(" den Fehlerc
     ode 10.\n\n");
 43:   PUT(HI_ON"\t                (c)M&T - ges
     chrieben von Daniel Görtz\n\n"HI_OFF);
 44: }
 45: void Hochkomma_beruecksichtigen(char **Bef
     ehle)
 46: {char *Befehl;
 47:   while(*Befehle)
 48:   { Befehl = *Befehle;
 49:     while(*Befehl)
 50:     { if(*Befehl == '\'' && *(Befehl+1) ==
      '\'')
 51:       { char *Dummy = Befehl;
 52:         while(*Befehl = *(Befehl+1)) Befeh
     l++;
 53:         Befehl = Dummy;
 54:       }
 55:       else
 56:         if(*Befehl == '\'') *Befehl = '\"'
     ;
 57:       Befehl++;
 58:     }
 59:     Befehle++;
 60:   }
 61: }
 62: enum Fehler Befehle_ausfuehren(char **Befe
     hle, struct FileHandle *OutputFile)
 63: { while(*Befehle)
 64:     if(!(Execute(*Befehle++, NULL, OutputF
     ile)))
 65:       return Befehl_Fehler;
 66:   return Kein_Fehler;
 67: }
 68: enum Fehler durchlaufe_Verzeichnis(char *V
     erzeichnis, char **Befehle, struct FileHan
     dle *OutputFile, enum wann Zeitpunkt)
 69: { enum Fehler Error = Kein_Fehler;
 70:   struct FileLock *lock = NULL, *oldlock =
      NULL;
 71:   struct FileInfoBlock *InfoBlock = NULL;
 72:   if(!(lock = (struct FileLock *) Lock(Ver
     zeichnis, ACCESS_READ)))
 73:       { Error = Disk_Fehler; goto FunkEnde;
      }
 74:   oldlock = (struct FileLock *) CurrentDir
     ((struct FileLock *) lock);
 75:   if(!(InfoBlock = (struct FileInfoBlock *
     )
 76:         malloc(sizeof(struct FileInfoBlock)
     )))
 77:       { Error = Speicher_Fehler; goto FunkE
     nde; }
 78:   if(!(Examine((BPTR) lock, (BPTR) InfoBlo
     ck)))
 79:       { Error = Disk_Fehler; goto FunkEnde;
      }
 80:   if(Zeitpunkt == Anfang)
 81:     Error = Befehle_ausfuehren(Befehle, Ou
     tputFile);
 82:   while(ExNext((BPTR) lock, (BPTR) InfoBlo
     ck) && Error == Kein_Fehler)
 83:   { if(InfoBlock->fib_DirEntryType >= 0)
 84:       durchlaufe_Verzeichnis(InfoBlock->fi
     b_FileName, Befehle, OutputFile, Zeitpunkt
     );
 85:     if(BREAK) { Error = Abbruch; goto Funk
     Ende; }
 86:   }
 87:   if(Zeitpunkt == Ende)
 88:     Error = Befehle_ausfuehren(Befehle, Ou
     tputFile);
 89:   FunkEnde:
 90:   if(lock)
 91:   { UnLock((BPTR) lock);
 92:     CurrentDir(oldlock);
 93:   }
 94:   if(InfoBlock) free(InfoBlock);
 95:   return Error;
 96: }
 97: void main(int argc, char **argv)
 98: { struct FileHandle *OutputFile;
 99:   enum wann Zeitpunkt = Anfang;
100:   enum Fehler Error;
101:   OutputFile = (struct FileHandle *) Outpu
     t();
102:   if(argc >= 3)
103:   { Hochkomma_beruecksichtigen(argv);
104:     if(!strcmp(argv[1], "-e") || !strcmp(a
     rgv[1], "-E"))
105:       Zeitpunkt = Ende;
106:     if(!strcmp(argv[1], "-a") || !strcmp(a
     rgv[1], "-A") || Zeitpunkt == Ende)
107:       Error = durchlaufe_Verzeichnis(argv[
     2], argv+3, OutputFile, Zeitpunkt);
108:     else
109:       Error = durchlaufe_Verzeichnis(argv[
     1], argv+2, OutputFile, Zeitpunkt);
110:   }
111:   else
112:     Error = Falscher_Aufruf;
113:   switch(Error)
114:   { case Speicher_Fehler:
115:       PUT("Fehler wegen Speichermangels au
     fgetreten !\n");
116:       exit(10);
117:     case Disk_Fehler:
118:       PUT("Verzeichnis nicht vorhanden ode
     r zerstört !\n");
119:       exit(10);
120:     case Befehl_Fehler:
121:       PUT("Befehl konnte nicht ordnungsgem
     äß ausgeführt werden !\n");
122:       exit(10);
123:     case Falscher_Aufruf:
124:       Aufruf(argv[0], OutputFile);
125:       exit(10);
126:     case Abbruch:
127:       PUT("*** BREAK\n");
128:       exit(10);
129:     case Kein_Fehler:
130:       exit(0);
131:   }
132: }                   © 1993 M&T
```

»All.c«: Weitet DOS-Befehle auf ganze Verzeichnisse aus

Gleitpunktzahlen

Genauigkeit zählt

Das Betriebssystem des Amiga unterstützt eine Reihe Datenformate unterschiedlicher Genauigkeit. Jedes bietet eine Reihe von Vor- und Nachteilen: Entweder man will es ganz genau, oder man möchte besonders schnell sein.

von Roger Fishlin

Der bekannteste numerische Datentyp der Klasse der Festpunktzahlen ist »Integer«, der nur ganze Zahlen speichert. Um Nachpunktstellen zu verarbeiten, muß ein anderer Datentyp mit Gleitpunktdarstellung gewählt werden. In PASCAL gibt es für reelle Zahlen den Typ REAL, analog in der Programmiersprache C den Typ FLOAT. Der Name FLOAT leitet sich aus der englischen Bezeichnung für Gleitpunktdarstellung ab: »Floating Point«. Bei sehr kleinen oder großen Beträgen ist die Exponentialdarstellung üblich: der Ausdruck »1.2E-20« bedeutet z.B. »$1.2*10^{-20}$«.

Die aktuelle Betriebssystemversion Kickstart 2.0 bietet dem Programmierer insgesamt drei Gleitpunktdarstellungen an:
- Fast Floating Point (FFP)
- Single Precision (IEEE-SP)
- Double Precision (IEEE-DP)

»Single Precision« wurde erst mit Kickstart 2.0 eingeführt, »Fast Floating Point« und »Double Precision« existieren bereits unter Kickstart 1.2. Die beiden IEEE-Formate wurden rechnerübergreifend als Standardimplementationen des »Institute for Electrical and Electronic Engineers«, kurz »IEEE«, definiert.

Das FFP-Format unterstützt der Amiga durch die beiden Libraries »mathffp.library« und »mathtrans.library«. Die erste Bibliothek ist ROM- und die zweite disk-resident, muß also bei Bedarf aus dem Verzeichnis »LIBS:« geladen werden. Die »mathffp.library« enthält Grundoperationen wie Addition und Multiplikation; transzendente Funktionen (z.B. Exponentionalfunktion und trigonometrische Funktionen wie Sinus) findet man in der »mathtrans.library«. Die maximale Genauigkeit beträgt im FFP-Format sieben Dezimalstellen. Außer der Null können Zahlen im Bereich von $+/-10^{-20}$ bis $+/-10^{18}$ dargestellt werden. Auf den ersten Blick mag es nicht einleuchtend sein, weshalb kein zusammenhängendes Intervall möglich ist. Der Grund: Die Darstellung der Zahlen im Intervall [0,10^-20] und dem analogen Bereich der negativen Zahlen würde mehr Bits als in der Norm vorgesehen benötigen.

Intern entspricht eine FFP-Zahl 32 Bit (ein Longword), arithmetische Operationen dürfen allerdings nur mittels Library-Routinen ausgeführt werden. Assembler-Programmierer können zum Beispiel zwei FFP-Zahlen nicht durch den Assembler-Befehl »ADD« addieren. Die Linkerbibliothek »amiga.lib« enthält u.a. zwei Routinen für die Umwandlungen eines ASCII-Strings in eine FFP-Zahl (»afp«) und umgekehrt (»fpa«). Nähere Informationen entnehmen Sie der offiziellen Dokumentation ([2]). Assembler-Programmierer müssen die Parameter in umgekehrter Reihenfolge auf dem Stack ablegen, »JSR _afp« bzw. »JSR _fpa« ausführen und den Stack-Pointer wieder korrigieren. Vergessen Sie nicht, das Objectfile anschließend mit dem »amiga.lib« zu linken.

Zwar ist auf einem Amiga mit 68000er CPU die Verarbeitung von Zahlen in FFP-Darstellung schneller als die nach der IEEE-Norm, aber bei einem mathematischen Coprozessor (MC68881/2) ändert sich die Relation. Der Grund: Die FFP-Libraries können nicht von Coprozessoren profitieren, da er im Gegensatz zur IEEE-Norm das FFP-Format nicht beherrscht. Alle IEEE-Libraries arbeiten hingegen (seit Kickstart 1.3 automatisch) mit dem Coprozessor. Fehlt er, müssen die Bibliotheken dessen Befehle mit Hilfe der CPU zeitaufwendig emulieren.

Ähnlich der FFP-Libraries sind die IEEE-Bibliotheken in arithmetische Grundoperationen und transzendente Funktionen unterteilt: »mathieeesingbas.library«, »mathieeesingtrans.library«, »mathieeedoubbas.library«, »athieeedoubtrans.library«. Bis auf die Library mit den Basisoperationen für einfache Genauigkeit sind momentan die Bibliotheken disk-resident.

Bei einfacher Genauigkeit (»Single Precision«) werden Gleitpunktzahlen ebenfalls in 32 Bit abgelegt. Jedoch unterscheidet sich der Aufbau vom FFP-Format, weshalb man die Zahlendarstellungen nicht direkt austauschen darf. Dargestellt werden können neben der Null Werte von +/- 10^-37 bis +/- 10^38. Arithmetische Operationen mit einfacher Genauigkeit erzielen bestenfalls auf sieben Dezimalstellen das korrekte Resultat.

In 64 Bit, also zwei Longwords, wird die Gleitpunktzahl bei doppelter Genauigkeit im IEEE-Format kodiert. Neben der Null können Zahlen im Bereich $+/- 10^{-308}$ bis $+/- 10^{307}$ dargestellt werden, ein Rechenergebnis ist höchstens auf 16 Dezimalstellen exakt. *ub*

Literatur:

[1] Commodore-Amiga, Inc. : "AMIGA ROM Kernal Reference Manual: Libraries", 1992, third Edition, Addison-Wesley Publishing Company, Inc., ISBN 0-201-56774-1

[2] Commodore-Amiga, Inc. : "AMIGA ROM Kernal Reference Manual: Includes And Autodocs", 1992, third Edition, Addison-Wesley Publishing Company, Inc., ISBN 0-201-56773-3

ARexx-Utilities

Königliche Hilfe

An Stelle von AmigaBASIC werden die neuen Amigas mit der Interpretersprache ARexx ausgeliefert. Was leistet diese Sprache? Was kann man mit Ihr anfangen? Hier ein paar Beispiele, die zeigen, daß ARexx Ihnen mit Ihrem Amiga viel abnehmen kann.

von Ilse u. Rudolf Wolf

ARexx wird meist nur zum Steuern oder als Ergänzung anderer Programme verwendet. Dabei wird von vielen häufig übersehen, daß diese Sprache – obwohl als Interpretersprache langsam – mehr Möglichkeiten bietet als AmigaDOS-Scripts und daher auch für Batch-Dateien und eigenständige Programme verwendet werden kann. Wenn zusätzlich Bibliotheken aus dem PD-Bereich eingebunden werden (z.B. die »rexxarp.librar« und »screenshare.library«), kann man sogar fast alle Möglichkeiten des Amiga-Betriebssystems nutzen. Für kurze Dienstprogramme eignet sich ARexx besonders. Das wollen wir Ihnen anhand einiger ARexx-Scripts zeigen.

DelInfo.rexx:
Wenn die Diskettensammlung wächst, verliert man bald die Übersicht, was, wo drauf ist. Als einfachste Möglichkeit zur Archivierung genügt es, das Inhaltsverzeichnis auszudrucken und abzulegen. Leider ist so ein Ausdruck unübersichtlich: Erstens wegen der ».info«-Dateien und zweitens weil zweispaltig. Das erste ARexx-Script, das wir vorstellen, »DelInfo.rexx«, macht einen Directory-Ausdruck übersichtlicher.

Die Ausgabe erfolgt einspaltig, wobei alle Dateien mit der Endung ».info« übersprungen werden. Das Ergebnis wird in der RAM-Disk als Text unter dem Namen »Dir.txt« abgelegt und anschließend mit MORE angezeigt, so daß ein Vor- und Zurückblättern im Inhaltsverzeichnis möglich wird.

```
/* ======== DelInfo.rexx ======== */
options prompt " Verzeichnis u. Option >"
pull dirname
befehl = "dir >ram:dummy" dirname
say " Bitte warten ..."
/* AmigaDOS-Befehl ausführen */
address COMMAND befehl
/* Dateien öffnen */
  open(lese,"ram:dummy","r")
  open(sende,"ram:Dir.txt","w")
/* Datum und Uhrzeit holen */
datum=DATE();dd=left(datum,2)
if left(dd,1)="0" then dd=right(dd,1)
/* Ausgabestring formatieren */
dt = DATE("W") dd DATE("M") right(datum,4)" -
                                    "TIME()
/* Überschrift */
writeln(sende,"Directory von" dirname)
writeln(sende,dt)
/* Umformatierungs-Schleife */
do while ~eof(lese)
 zeile= readln(lese)
 y=" ";n=0
 /* Leerzeichen für ermitteln */
 do while y=" " & n< length(zeile)
  n=n+1
  y=substr(zeile,n,1)
  end
 if n~=0 then n=n-2
  y=copies(" ",n)
 /* Zeilen-Parser */
 parse var zeile eins zwei
 zwei=strip(zwei,"b")
  if zwei="(dir)" then do
    eins= eins || zwei
    writeln(sende,y || eins)
    end
  else do
  if index(eins,'.info')=0 & length(eins)>0
    then writeln(sende,y || eins)
  if index(zwei,'.info')=0 & length(zwei)>0
    then writeln(sende,y || zwei)
  end
end
/* Geöffnete Dateien schließen */
call close(lese)
call close(sende)
/* Hilfs-Datei schließen */
address COMMAND
'delete >NIL: ram:dummy'
'echo "*ec"'
'more ram:Dir.txt'                © 1993 M&T
```

»DelInfo.rexx«: gibt ein Verzeichnis aus wie DIR – allerdings ohne Info-Dateien

Die Bedienung ist einfach; es braucht nur der Directory-Befehl eingetippt werden,z.B.:

`df1:`	**gibt nur das Hauptverzeichnis der Diskette im Laufwerk DF1 aus;**
`df1: all`	**gibt alle Verzeichnisse der Diskette im Laufwerk DF1 aus;**
`sys:prefs`	**zeigt den Inhalt von »Prefs« von der System-Diskette;**
`c:`	**listet den Inhalt von C:.**

aList.rexx
Für die Archivierung einer Diskette ist es sinnvoll, auch die Dateilängen und das Erstellungsdatum festzuhalten. Für ein Verzeichnis liefert diese Angaben der LIST-Befehl. Doch den Inhalt aller Verzeichnisse der Reihe nach mit LIST auszugeben, ist eine aufwendige Sache. Mit dem ARexx-Script »aList.rexx« ist alles wesentlich einfacher, denn man braucht nur das Laufwerk (z.B.: df0:, df1:, dh0: etc.) einzugeben. Alles andere besorgt »aList.rexx«.

Auch hier werden alle Dateien mit der Endung ».info« übersprungen. Das Ergebnis wird in der RAM-Disk als Textdatei unter dem Namen »aList.txt« abgelegt. Auch hier erfolgt anschließend die Anzeige des Ergebnisses mit MORE.

```
/* ========== aList.rexx ========== */
options prompt " Welches Laufwerk? > "
pull drive
say " Bitte warten..."
/* Header speichern */
befehl='list >ram:dummy' drive
                        'dirs quick nohead'
address COMMAND befehl
/* Inhalte speichern */
open(lese,"ram:dummy","r") /* Hilfs-Datei */
open(sende,"ram:aList.txt","w")
call close(sende) /* LIST schreibt in Datei */
do while ~eof(lese) /*.info-Datei auslassen */
 zeile=readln(lese)
 zeile='list >>ram:aList.txt' drive||
                       zeile 'p ~(#?.info)'
 address COMMAND zeile
  open(leer,"ram:aList.txt","a")
   writeln(leer,"") /* Leerzeile */
  call close(leer)
end
call close(lese)

address COMMAND
'echo "*ec"'
'more ram:aList.txt' /* Ausgabe */
'delete >NIL: ram:dummy'          © 1993 M&T
```

»List.rexx«: wenn Sie den Inhalt einer ganzen Diskette auflisten möchten

Copy600HD.rexx
Der Amiga600HD wird mit einer Festplatte und einem Diskettenlaufwerk geliefert. Das bedingt, daß man beim Kopieren einer Diskette mit nur einem Laufwerk zum Disk-Jockey wird. Abhilfe schafft das ARexx-Script »Copy600HD.rexx«.

```
/*          Copy600HD.rexx           */
/* kopiert Disk  mit A600HD in einem Durchgang
*/
say ""
say " Bitte Quell-Diskette einlegen"
options prompt " und RETURN-Taste drücken"
pull warte
address COMMAND
/* Puffer anlegen */
if ~exists('dh0:buffer') then do
  'makedir dh0:Buffer'
  bufnam="dh0:Buffer"
  end
 empty=bufnam||"/#?"
 'delete >NIL:' empty
 target="df0: to" bufnam "all"
 'copy' target
 start:
 'echo "*ec"'
 say "Bitte Ziel-Diskette einlegen"
 options prompt " und RETURN-Taste drücken"
 pull warte
 target=bufnam "to df0: all"
 'copy' target
 'echo "*ec"'
 say " Diskette kopiert!"
 options prompt " Noch eine Diskette?-j/n > "
 pull jn
 if upper(jn)~="J" then do
  say " Puffer wird gelöscht!"
  'delete >NIL:' bufnam 'all'
  end
  else do
   signal start
   end               © 1993 M&T
```

»Copy600HD.rexx«: hilft, wenn Sie mit dem Amiga 600 HD Diskette kopieren

Das Programm legt auf der Festplatte einen Puffer an und kopiert die Quelldiskette vom Laufwerk DF0: in den Puffer. Danach erscheint die Aufforderung, die Zieldiskette einzulegen, und der Kopiervorgang erfolgt in umgekehrter Richtung. Ist er beendet, fragt der Amiga, ob der Puffer auf eine weitere Diskette kopiert werden soll – eine zeitsparende Methode für Mehrfachkopien.

DelRemRX.rexx und DelRemBas.rexx

Ein Listing sollte mit Kommentaren versehen werden, weil es dadurch verständlicher wird. In Interpreter-Sprachen verlangsamt das allerdings die Ablaufgeschwindigkeit. Es ist daher sinnvoll, eine kommentarlose Arbeitskopie zu verwenden.

»DelRemRX.rexx« entfernt aus einem ARexx-Script alle Kommentare, wobei die erste Zeile – die ja in ARexx eine Kommentarzeile sein muß – erhalten bleibt.

```
/* ======= DelRem.rexx ======= */
say " Entfernt Kommentare aus ARexx-Programmen"
say ""
options prompt " Quell-Datei: "
pull source
options prompt " Ziel-Datei : "
pull target
open(lese,source,"r")
open(sende,target,"w")
/* ARexx-Kennung uebernehmen */
zeile= readln(lese)
writeln(sende,zeile)
do while ~eof(lese)
 zeile= readln(lese)
 x1=index(zeile,"/*",1)
 y1=index(zeile,"*/")
  select
    when zeile=""then NOP
    when x1~=0
     then zeile=delstr(zeile,x1)
    when y1~=0 & x1=0
     then zeile=delstr(zeile,1)
   otherwise NOP
  end
 zeile=trim(zeile)
 if zeile ~=""
  then writeln(sende,zeile)
end
call close(lese)
call close(sende)
say "READY!"
```

© 1993 M&T

»DelRem.Rexx«: entfernt Kommentare aus einem ARexx-Script

Analog dazu entfernt das Programm »DelRemBAS.rexx« alle Kommentare aus einem AmigaBASIC-Listing, wobei wieder die erste Zeile erhalten bleibt.

Die Bedienerführung beider Dienstprogramme ist selbsterklärend.

```
/* ======= DelRem.rexx ======= */
say " Entfernt Kommentare aus AmigaBASIC-
Programm"
say ""
options prompt " Quell-Datei: "
pull source
options prompt " Ziel-Datei : "
pull target
open(lese,source,"r")
open(sende,target,"w")
/* 1.Zeile übernehmen */
zeile= readln(lese)
writeln(sende,zeile)
/* Kommentare entfernen */
do while ~eof(lese)
 zeile= readln(lese)
 x1=index(zeile,"REM",1)
 x2=index(zeile,"'")
 x3=index(zeile,":REM",3)
  select
    when zeile=""then NOP
    when x3~=0
     then zeile=delstr(zeile,x3)
    when x1~=0
     then zeile=delstr(zeile,x1)
    when x2~=0
     then zeile=delstr(zeile,x2)
   otherwise NOP
  end
 zeile=trim(zeile)
 if zeile ~=""
  then writeln(sende,zeile)
end
call close(lese)
call close(sende)
say "READY!"
```

© 1993 M&T

»DelRemBAS.Rexx«: entfernt alle Kommentare aus einem BASIC-Listing

Guten START mit ARexx

Damit Sie die ARexx-Scripts auch von der Workbench aus starten können, müssen Sie fünf Project-Icons bereitstellen (z.B. mit dem IconEdit). Diese erhalten jeweils den Namen des ARexx-Scripts. In jedes Icon wird in das »Default Tool«-Feld eingetragen:

`SYS:rexxc/RX`

Die »Tool Types« unterstützen als Argumente CONSOLE und CMD, wobei CONSOLE für ein Fenster steht. Wenn Sie dort

`CONSOLE=CON:0/0/639/255/ARexx Output`

eintragen, erhalten Sie ein bildschirmfüllendes Fenster.

Der Vollständigkeit halber: CMD ist das »Tool Types«-Schlüsselwort für einen Kommandostring. Wird dort nichts eingetragen, versucht der RX-Befehl, die zum Icon gehörende Datei als ARexx-Programm auszuführen. Für die vorgestellten Programme ist ein CMD-Eintrag nicht erforderlich. CMD wird jedoch gebraucht, wenn einem ARexx-Script Argumente mitgegeben werden müssen oder wenn das Project-Icon nur als Starter für ein ARexx-Programm dient. Ein analoges Beispiel ist das Shell-Icon.

Abschließend noch ein Hinweis: Für den Workbench-Start muß der ARexx-Server bereits aktiviert sein. Wenn nicht, gibt es folgende Fehlermeldung:

```
rexxmast:unknown command
rexxmast failed returncode 10
ARexx server not active
```

Da nun alles klappen dürfte, bleibt nur noch, Ihnen viel Spass mit ARexx zu wünschen. *ub*

Literaturnachweis:
Eric Giguére: AMIGA Programmer's Guide to ARexx
Commodore-Amiga,Inc.,West Chester, Pennsylvania
Commodore: Handbuch zur Systemsoftware 2.0

APIG-Library (Fish 634)

Der König ist tot - Es lebe APIG

Mit der im vorigen Artikel vorgestellten Interpretersprache ARexx erhebt sich vor allem eine Frage: Kann ARexx auch AmigaBASIC ersetzen? Die Antwort lautet eindeutig »ja« – vor allem, wenn zusätzlich eine leistungsfähige externe Funktionsbibliothek wie APIG zur Verfügung steht.

Ilse u. Rudolf Wolf

Wie auf Seite 55 erwähnt werden alle Amiga-Modelle ab Betriebssystemversion (OS) 2.0 nicht mehr mit AmigaBASIC ausgeliefert. Dafür ist ARexx ein fester Bestandteil des Amiga-Betriebssystems. Was diese Sprache kann, haben wir im genannten Artikel beleuchtet. Doch ARexx leistet noch einiges mehr als Skriptsprachen.

Grundsätzlich ist ARexx eine Kommandosprache, mit der es möglich ist, Programme fernzusteuern. In der Literatur findet man daher fast nur solche Anwendungen.

ARexx kann aber auch als die logische Weiterentwicklung des CLI betrachtet werden, denn im Gegensatz zum CLI besitzt es einen komfortablen Befehlssatz. Dazu kommt, daß dieser mit externen Funktionsbibliotheken fast unbegrenzt erweiterbar ist. Aus dem Fish-Aquarium kann man sich eine Reihe brauchbarer Funktionsbibliotheken herausangeln (siehe Tabelle).

Fish-Disk	*Name*	*Funktion*
227	RexxArpLib	Zugriff auf ARP-Funktionen ermöglichen.
227	RexxMathLib	ARexx um trigonometrische Funktionen erweitern.
308	ScreenshareLib	von RexxArpLib benötigt
459	RxGen	Zugriff auf Amiga-Bibliotheken ermöglichen
463	RexxIntuition	Zugriff auf diverse Intuition-Funktionen ermöglichen.
629	RexxRMF	Datenbank in ARexx erstellen (AVL trees)
634	APIG	ermöglicht Zugriff auf Exec-, Asl-, Graphics-, Intuition-, Layers-, Utility- und Gadtools- Funktionen.
682	RexxHostLib	Bibliothek mit Funktionen für eine ARexx-Schnittstelle

Tabelle: Im PD-Pool finden Sie viele Bibliotheken, um ARexx zu erweitern

Damit ist ARexx nicht nur als eine Batch-Sprache zur automatisierten Steuerung oder eine Makro-Sprache zur Steuerung eines Anwenderprogrammes sondern als universelle Programmiersprache einsetzbar und übertrifft AmigaBASIC in vielen Belangen.

Die »apig.library«

Der beste Fang aus dem Fish-Aquarium dürfte derzeit die APIG (A Programmers Intuition & Graphics Library) auf der Fish 634 sein, denn mit der »apig.library« ist der Zugriff auf 288 Funktionen aus den Amiga-OS2.0-Bibliotheken (Asl, Exec, Gadtools, Graphics, Intuition, Layers und Utility) möglich. Zusätzlich enthält die Bibliothek über 600 Konstanten und Flags, wie sie in den Include-Files definiert sind. Kein Wunder, daß die »apig.library« 61220 Byte lang ist und daß das auf Diskette befindliche, ausgedruckte Manual rund 50 Seiten umfaßt.

```
 1: /* ============ Flush.rexx ===========*/
 2: say '0a'x "ARexx-Resources-Liste:"
 3: say "" show('l')
 4: if(show('l','rexxsupport.library')) then do
 5:   say " REMLIB -> rexxsupport.library"
 6:   call remlib('rexxsupport.library')
 7:   end
 8: if(show('l','apig.library')) then do
 9:   say " REMLIB -> apig.library"
10:   call remlib('apig.library')
11:   end
12: if(show('l','rexxmathlib.library')) then do
13:   say "REMLIB -> rexxmathlib.library"
14:   call remlib('rexxmathlib.library')
15:   end
16: address COMMAND 'avail >NIL: flush'
17: say '0a'x "Neue ARexx-Resources-Liste:"
18: say "" show('l') '0a'x
19: say " Ferner wurden auch alle ungenutzen"
20: say " LIBS u. DEVS aus dem Speicher entfern
    t"
21: options prompt " RETURN-Taste beendet.."
22: pull holdit         © 1993 M&T
```

»Flush.rexx«: Ein erstes Beispiel für den Einsatz von APIG

Das kurze Listing »Flush.rexx« zeigt, wie ein Programm aussieht, das mit Hilfe der »apig.library« auf Systemfunktionen zurückgreift. In diesem Fall löscht das Programm ungenutzte Libraries und Devices aus dem Speicher.

In den meisten Fällen konvertiert die »apig.library« die ARexx-Stringparameter in ein Format, das die Amiga-Bibliotheksfunktionen erwarten. C-Programmierer werden ihre Freude daran haben, denn die Portierung ermöglicht ein flexibles und funktionelles Programmieren ähnlich wie in C.

Folgende Strukturen werden unterstützt: Menu, MenuItem, SubItem, Requesters, Boolean, String- und Proportional-Gadgets, Borders, IntuiText, 16-Bit-Arrays, Layers und IFF (via »iff.library« von Christian Weber). Allen die mit Intuition/Graphics-Funktionen und Strukturen nicht vertraut sind, empfiehlt der Programmierer Ronnie E. Kelly daher das Studium der Includes & Autodocs.

Schauen wir uns ein zweites Beispiel an: »AslFile1.rexx« demonstriert, wie man einen ASL-Requester von ARexx aus aufruft.

```
 1: /* ==========  AslFile1.rexx ==========
 2:     Einfacher MultiFileRequester mit
 3:     ALLOCFILEREQUEST() u. REQUESTFILE() */
 4: /* Externe Bibliotheken einbinden */
 5: if(~show('l','rexxsupport.library'))
 6: then call addlib('rexxsupport.library',0,-30,0)
 7: if(~show('l','apig.library'))
 8: then call addlib('apig.library',0,-30,0)
 9: /* Requester-Struktur anlegen */
10:  freq = ALLOCFILEREQUEST()
11: /* Parameter für REQUESTFILE */
12:  multi = 1 /* Mehrfachauswahl möglich */
13:  save = 0
14:  hail = "MultiFileRequester"
15:  dir  = "RAM:" /* voreingest. Verzeichnis */
16:  file = ""  /* voreingest. Filename */
17:  pat  = 1   /* Wenn pat > 0 :Pattern-Gadget */
18:  nofile = 0 /* nofile > 0: nur Verzeichnisse*/
19:  win = null() /* Zeiger auf Parentwindow */
20:  x   = 0      /* linke obere Ecke */
21:  y   = 0
22:  wid = 500    /* Requester-Breite */
23:  hgt = 190    /* Requester-Höhe  */
24:  sep = '0a20'x /* Separator Mehrfachauswahl */
25: /* Requester aufrufen */
26: filename = REQUESTFILE(freq,multi,save,hail,d
    ir,file,pat,nofile,win,x,y,wid,hgt,sep)
27: /* Ergebnis auswerten */
28: if filename = null()
29:    then say "Cancel angeklickt!"
30:    else do
31:      say " Angeklickt wurde:" '0a'x
32:      say "" filename
33:      end
34: /* Aufräumen */
35: call FREEFILEREQUEST(freq)
36: /* call remlib('rexxsupport.library')
37:    call remlib('apig.library')
38:    address COMMAND 'avail >NIL: flush' */
    © 1993 M&T
```

»AslFile1.rexx«: ASL-Requester mit der »apig.library« aus ARexx aufrufen

Wie alle Bibliotheken gehört die »apig.library« ins LIBS-Verzeichnis. Falls IFF-Funktionen verwendet werden, auch die »iff.library« (befindet sich ebenfalls auf der Fish 634); und wenn trigonometrische Funktionen gebraucht werden, auch die »rexxmath.library« von der Fish-Disk 227.

Um für alle Fälle gerüstet zu sein und damit Sie möglichst viele unterschiedliche Funktionen in Ihren ARexx-Programmen

nutzen können, sollten in LIBS: folgende Bibliotheken vorhanden sein:

```
- apig.library          Fish 634
- iff.library           Fish 634
- rexxmathlib.library   Fish 227
- rexxsupport.library
- rexxsyslib.library
```

Startvorbereitungen

Bevor man APIG-Funktionen aufrufen kann, muß die »apig.library« in die ARexx-internen Bibliothekenliste aufgenommen werden. Kelly macht das in seinen Programmbeispielen mit

```
call addlib('apig.library',0,-30,0)
```

oder mit:

```
x = addlib('apig.library',0,-30,0)
```

Er setzt dabei stillschweigend voraus, daß die »rexxsupport.library« bereits geladen wurde. Ist das nicht der Fall, brechen viele seiner Beispiele mit einer Fehlermeldung ab, weil sie Funktionen aus dieser Bibliothek enthalten. Wir empfehlen daher, den ADDLIB-Aufruf durch die folgende Sequenz zu ersetzen:

```
if(~show('l','rexxsupport.library'))
 then call addlib('rexxsupport.library',
                                    0,-30,0)
 if(~show('l','apig.library'))
  then call addlib('apig.library',0,-30,0)
```

Hier wird abgefragt, ob die Bibliotheken in die Liste eingebunden sind. Wenn nicht, werden sie in den Speicher geladen und in die Liste aufgenommen.

Es gehört zu einem sauberen Programmierstil, die externen Bibliotheken am Programmende wieder aus der Liste zu entfernen. Die folgenden beiden Zeilen erledigen das:

```
call remlib('rexxsupport.library')
call remlib('apig.library')
```

Damit sind die Libraries aus der ARexx-Liste entfernt, bleiben aber im RAM. Wenn Speicherknappheit herrscht, kann man den belegten Speicher mit dieser Zeile freigeben:

```
address COMMAND 'avail >NIL: flush'
```

Kelly verwendet in vielen Beispielen als Warte-Befehl das Schlüsselwort »wait« (z.B.: wait 5 secs). ARexx kennt das nicht und interpretiert das natürlich nicht als Befehl. In den Beispielen müssen daher alle Wait-Klauseln ersetzt werden. Beispiel:

```
wait 5 secs
```

durch

```
address COMMAND 'wait 5 secs'
```

Wenn die »rexxsupport.library« geladen ist, geht´s kürzer mit:

```
call delay(250)
```

Weil DELAY als Argument die Ticks/sek verlangt, muß für 5 Sekunden der Wert 250 eingesetzt werden.

Diese Hinweise sollten Ihnen helfen, Enttäuschungen mit dieser großartigen Bibliothek zu vermeiden.

Anwendungsbeispiele

Wir haben für Sie einige Anwendungsbeispiele programmiert. Zum einen das Listing »Aerea.rexx«, das die Area-Funktionen der «graphics.library« nutzt, und »iff.rexx«, mit dem Sie über ARexx mit der »apig«- und der »iff.library« IFF-Dateien laden können.

Damit Sie die Listings nicht abtippen müssen, sind sie auch auf der Begleitdiskette enthalten (siehe Seite 114). Zusätzlich finden Sie auf der Diskette noch einige weitere Beispiele, um ASL-Requester aufzurufen, Fonts einzustellen usw.

Alle Listings sind reichlich kommentiert. Damit Sie sich besser zurechtfinden, folgt eine ausführliche Beschreibung der Parameterübergabe an die Funktionen OPENSCREEN, OPENWINDOW und PITEXT, die in den meisten Listings vorkommen:

```
OPENSCREEN(left,top,width,height,depth,
           dpen,bpen,vmodes,type,title)
```

Diese Funktion öffnet einen Custom-Screen. Jeder Aufruf erzeugt einen Prozeß mit Namen »apig.screen.N«. N ist eine sequentielle Nummer zur Identifikation, die automatisch vergeben wird. Der Port kann mit dem Prozeßnamen angesprochen werden. Derzeit schließt aber jede Message den Screen. Die Bedeutung der Parameter für OPENSCREEN ist folgende:

```
left   - x-Position der linken oberen Ecke
top    - y-Position der linken oberen Ecke
width  - Breite
height - Höhe
depth  - Tiefe (Anz. d. Bitplanes)
dpen   - Vordergrundfarbe
bpen   - Hintergrundfarbe
vmodes - Darstellungsmodus: HIRES etc.
type   - Typ: CUSTOMSCREEN,
              WBENCHSCREEN etc.
title  - Titel (String)

Rückgabewert: Zeiger auf den Screen als
              Rexx-Hexstring oder ein Null-
              Hexstring ('0000 0000'x),
              wenn OPEN mißlungen ist.
```

Die Parameter für OPENWINDOW sind noch umfangreicher. Der Aufruf lautet:

```
OPENWINDOW(port,left,top,wid,hgt,dpen,bpen,
           IDCMP,flags, title,scr,console,
           bitmap,chkmark,gadlist)
```

Die Parameter im einzelnen:

```
port  - Name des IDCMP-Messageports, an
        den Nachrichten gesendet werden.
left  - x-Position der linken oberen Ecke
top   - y-Position der linken oberen Ecke
wid   - Breite
hgt   - Höhe
dpen  - Vordergrundfarbe
bpen  - Hintergrundfarbe

IDCMP - IDCMP-Flags. Auswertung:
        class     = GETARG(msg,0)
        code      = GETARG(msg,1)
        qualifier = GETARG(msg,2)
        mousex    = GETARG(msg,3)
        mousey    = GETARG(msg,4)
        seconds   = GETARG(msg,5)
        micros    = GETARG(msg,6)
        window    = GETARG(msg,7)
        iaddress  = GETARG(msg,8)
        gadgetid  = GETARG(msg,9)
flags   - Window-Flags
title   - Titel (String)
scr     - Zeiger auf Screen, für Workbench-
          Screen null()
console - Parameter bestimmt, ob Fenster
          eine 'console' erhält (jeder Wert
          ungleich Null). Ermöglicht, mit
          WRITECONSOLE()-Funktion, in das
          Fenster zu schreiben.
bitmap  - Zeiger auf Bitmap, wenn Fenster
          ein Superbitmap-Window ist.
chkmark - (Normal 0) Zeiger auf ein
          Menü-Haken-Image
gadlist - Zeiger auf eine Gadget-Liste.
          Im Normalfall 0.

Rückgabewert: Zeiger auf Window als Rexx-
              Hexstring oder Null-Hexstring
              ('0000 0000'x), wenn OPEN
              mißlungen ist.
```

Mit den Funktionen CLOSESCREEN und CLOSEWINDOW schließt man ein Fenster wieder. Die Aufrufe lauten:

```
CLOSESCREEN(screen) - Schließt einen Screen
CLOSEWINDOW(window) - Schließt ein Window
```

Den Funktionen wird einfach der Zeiger auf die betreffenden Screen- bzw. -Fenstersturktur des zu schließenden Objekts übergeben.

Alt letztes ist noch die Funktion PITEXT zu erwähnen, mit der man einen Text ausgibt. Der Aufruf lautet:

```
PITEXT(rp,left,top,text,fp,bp,dm,font)
```

Man übergibt folgende Werte:

```
rp    - Zeiger auf den Rastport in den
        der Text ausgegeben wird.
left  - x-Offset in Punkten im Rastport
top   - y-Offset in Punkten im Rastport
text  - Textstring
fp    - Textfarbe
bp    - Hintergrundfarbe
dm    - Drawmode (JAM1,JAM2...)
font  - Zeiger auf die Textattribut-Struk-
        tur des Zeichensatzes
        (von MAKETATTR() ) oder Null
```

An die in unseren den Beispielen verwendeten Funktionen OPENSCREENTAGLIST, OPENWINDOWTAGLIST, ALLOCTAGITEMS werden die Parameter mit Tag-Listen übergeben. Deren Aufbau ist aus den Listings ersichtlich, speziell aus »AslFont.rexx« (auf der Diskette zum Heft).

Im folgenden finden Sie nun zwei Beispiele für den Einsatz der »apig.library«:

❒ »LoadIFF.rexx« lädt ein Bild, das in Form einer IFF-Datei vorliegt;

❒ »Aerea.rexx« demonstriert den Einsatz der Area-Funktionen.

Auf der Diskette zum Heft (siehe Seite 114) finden Sie weitere Beispiele sowie die komplette »apig.library« inklusive Dokumentation. Testen Sie, was man alles mit der Bibliothek machen kann. *ub*

Literaturnachweis:
Commodore-Amiga,Inc.,West Chester, Pennsylvania:
Eric Giguére - AMIGA Programmer's Guide to ARexx
Commodore: Handbuch zur Systemsoftware 2.0
Commodore: Includes & Autodocs V39.108

Fish-Disk 634: APIG
Fish-Disk 227: rexxmathlib.library

```
 1: /* ======== LoadIFF.rexx ========= */
 2:
 3: /* Externe Bibliotheken einbinden */
 4: if(~show('l','rexxsupport.library'))
 5:   then call addlib('rexxsupport.library',0,
    -30,0)
 6: if(~show('l','apig.library'))
 7:   then call addlib('apig.library',0,-30,0)
 8:
 9: /* Intuition-Konstanten und Flags setzen */
10: call set_apig_globals()
11:
12: /* Speicher für Requesterstruktur */
13: freq = ALLOCFILEREQUEST()
14:
15: /* Requester aufrufen */
16:  hailing = "Bild-Datei oder Cancel anklicke
    n.."
17: filename = REQUESTFILE(freq,,,hailing,,,,,
    ,20,400,220)
18: if filename = null() then do
19:   say " Cancel angeklickt!"
20:   call cleanup()
21:   exit
22:   end
23:
24: /* Selektierte IFF-Datei laden */
25: scr = 0
26: picture   = loadiff(filename)
27: if picture = '0000 0000'x then do
28:   say " Das ist kein IFF-ILBM- File!"
29:   call cleanup()
30:   exit
31:   end
32: width    = iffwidth(picture) /* Display-Par
    ameter */
33: height   = iffheight(picture)
34: depth    = iffdepth(picture)
35: viewmode = iffviewmode(picture)
36: ncolors  = iffcolors(picture)
37: colors   = iffcolortab(picture)
38: /* Screen öffnen */
39: scr   = openscreen(0,0,width,height,depth,1
    ,0,viemode,CUSTOMSCREEN,0)
40: scrrp = getscreenrastport(scr)
41: z     = useiffcolor(picture,scr)
42:
43: /* Image in den Screen-Rastport kopieren */
44: z = bltbitmaprastport(picture,0,0,scrrp,0,0
    ,width,height,c2d('00c0'x))
45:
46: pull holdit    /* Auf das Drücken der RETUR
    N-Taste warten */
47: call CLOSESCREEN(scr)
48: call FREEBITMAP(picture)
49: call cleanup() /* Aufräumen */
50: exit
51:
52: cleanup:
53:  call FREEFILEREQUEST(freq)
54:  /*
55:  call remlib('rexxsupport.library')
56:  call remlib('apig.library')
57:  address COMMAND 'avail >NIL: flush'
58:  */
59: return          © 1993 M&T
```

»LoadIFF.rexx« Lädt ein Bild, das als IFF-Datei vorliegt

```
 1: /* ============= Aerea.rex ============= */
 2: /*      Beispiele für Area-Funktionen      */
 3:
 4: /* Externe Bibliotheken einbinden */
 5: if(~show('l','rexxsupport.library'))
 6:   then call addlib('rexxsupport.library',0
    ,-30,0)
 7: if(~show('l','apig.library'))
 8:   then call addlib('apig.library',0,-30,0)
 9:
10: /* Intuition-Konstanten u. Flags initialis
    ieren */
11: call set_apig_globals()
12:
13: portname = "msgport"
14: p = openport(portname)
15:
16: scrtitle = " apig.screen"
17: wintitle = " apig.window"
18: winidcmp = CLOSEWINDOW
19: winflags = WINDOWCLOSE+WINDOWDRAG+WINDOWSI
    ZING+WINDOWDEPTH+GIMMEZEROZERO
20:
21: scr = openscreen(0,0,640,512,3,2,1,LACE+HI
    RES,CUSTOMSCREEN,scrtitle)
22: win  = openwindow(portname,0,12,640,400,1,
    0,winidcmp,winflags,wintitle,scr,0,0,0)
23: call activatewindow(win)
24: winrastp = getwindowrastport(win)
25: /* 4 Bytes für Füllmuster reservieren */
26: pat = allocmem(4,MEMF_CLEAR)
27: /* Füllmuster definieren */
28: z   = export(pat,'5555 aaaa'x,4)
29: /* Füllmuster setzen */
30: z   = setafpt(winrastp,pat,1)
31: /* Window-Rastport für area fills */
32: z = makearea(win,640,400,3000)
33: if z = 0 then call cleanup()
34: y=30
35: do twice = 1 to 2
36:  if twice = 2 then do
37:     z = setopen(winrastp,1) /* OutlinePen
    setzen */
38:     y=y+60
39:  end
40:  pen = 1
41:   /* Quadrate zeichnen */
42:  do x = 10 to 600 by 60
43:     if pen > 7 then pen = 1
44:     z = setapen(winrastp,pen)
45:     z = areamove(winrastp,x,y)
46:     z = areadraw(winrastp,x+50,y)
47:     z = areadraw(winrastp,x+50,y+50)
48:     z = areadraw(winrastp,x,y+50)
49:     z = areadraw(winrastp,x,y)
50:     z = areaend(winrastp)
51:     pen = pen + 1
52:  end
53:  z = setopen(winrastp,0)
54: end
55:
56: z = freearea(win)
57: z = makearea(win,640,400,3000)
58: /* Kreis */
59: z = export(pat,'0101 0101'x,4)
60: z = setdrmd(winrastp,JAM2)
61: z = setapen(winrastp,3)
62: z = setbpen(winrastp,4)
63: z = setopen(winrastp,1)
64: z = setafpt(winrastp,pat,0)
65: z = move(winrastp,420,140)
66: z = areacircle(winrastp,460,160,60)
67: z = areaend(winrastp)
68: /* Ellipse */
69: z = export(pat,'ffff ffff'x,4)
70: z = setapen(winrastp,5)
71: z = setbpen(winrastp,6)
72: z = setopen(winrastp,4)
73: z = setafpt(winrastp,pat,1)
74: z = areaellipse(winrastp,420,220,60,40)
75: z = areaend(winrastp)
76: z = setbpen(winrastp,0)
77:
78: /* Bborder-Array anlegen */
79: /* (20 bytes, 5points * 2x * 2y) */
80: barray = allocmem(5*4,MEMF_CLEAR)
81: x = setx(barray,0,0)   ; y = sety(bar-
ray,0,0)
82: x = setx(barray,1,150) ; y = sety(bar-
ray,1,0)
83: x = setx(barray,2,150) ; y = sety(barray,2
    ,100)
84: x = setx(barray,3,0)   ; y = sety(barray,3
    ,100)
85: x = setx(barray,4,0)   ; y = sety(bar-
ray,4,0)
86:
87: border1 = makeborder(win,barray,5,40,20,1,
    0,JAM2,0)
88: z = drawborder(winrastp,border1,0,220)
89: z = pitext(winrastp,200,270," <--- FLOOD i
    n 2 Sekunden ",1,2,JAM2,0)
90: call delay(100)
91:
92: z = setopen(winrastp,1)          /* OPen wie
    Border-Ccolor    */
93: z = flood(winrastp,0,60,250)     /* Punkt in
    nerhalb der Border */
94:
95: z = pitext(winrastp,200,300,"Ende in 10 Se
    kunden ",2,1,JAM2,0)
96: call delay(500)
97: z = freearea(win)  /* Speicher freigeben */
98: cleanup:
99:   z = freemem(pat,4)
100:  z = closewindow(win)
101:  z = closescreen(scr)
102: /*  call remlib('rexxsupport.library')
103: call remlib('apig.library')
104: address COMMAND 'avail >NIL: flush'*/
     © 1993 M&T
```

»Aerea.rexx«: Zeichenfunktionen kann man auch von ARexx aus einsetzen

IMPRESSUM

Chefredakteur: Albert Absmeier : – verantwortlich für den redaktionellen Teil
Stellv. Chefredakteur: Ulrich Brieden (ub)
Textchef: Jens Maasberg
Redaktion: Rainer Zeitler (rz)
freie Mitarbeiter: Ilse und Rudolf Wolf
Redaktionsassistenz: Catharina Winter, Helga Weber

So erreichen Sie die Redaktion:
Tel. 0 89/46 13-4 14, Telefax: 0 89/46 13-4 33
Hotline Do, 15-17.oo Uhr

Manuskripteinsendungen: Manuskripte und Programmlistings werden gerne von der Redaktion angenommen. Sie müssen frei sein von Rechten Dritter. Sollten sie an anderer Stelle zur Veröffentlichung oder gewerblichen Nutzung angeboten worden sein, muß das angegeben werden. Mit der Einsendung von Manuskripten und Listings gibt der Verfasser die Zustimmung zum Abdruck in der von Markt & Technik Verlag AG herausgegebenen Publikationen und zur Vervielfältigung der Programmlistings auf Datenträgern. Mit Einsendung von Bauanleitungen gibt der Einsender die Zustimmung zum Abdruck in von Markt & Technik Verlag AG verlegten Publikationen und dazu, daß die Markt & Technik Verlag AG Geräte und Bauteile nach der Bauanleitung herstellen läßt und vertreibt oder durch Dritte vertreiben läßt. Honorare nach Vereinbarung. Für unverlangt eingesandte Manuskripte und Listings wird keine Haftung übernommen.

Layout: Ulrich Brieden, Rainer Zeitler
Desktop Publishing: Ulrich Brieden, Rainer Zeitler
Titelgestaltung: Ulrich Brieden, Rainer Zeitler
Anzeigenleitung: Peter Kusterer
Anzeigenverwaltung und Disposition: Anja Böhl (233)

So erreichen Sie die Anzeigenabteilung:
Tel. 0 89/46 13-9 62, Telefax: 0 89/46 13-394

Leiter Vertriebsmarketing: Benno Gaab (740)
Vertrieb Handel: MZV, Moderner Zeitschriftenvertrieb GmbH & Co KG, Breslauer Straße 5 , Postfach 11 23, 8057 Eching, Tel. 0 89/31 90 06-0
Erscheinungsweise: Das Sonderheft Faszination Programmieren ist zunächst zweimal jährlich geplant
Bezugspreise: Das Einzelheft kostet DM 12.–.
Leitung Technik: Wolfgang Meyer (887)
Druck: L.N.Schaffrath, Grafischer Betrieb, 4170 Geldern 1
Warenzeichen: Diese Zeitschrift steht weder direkt noch indirekt mit Commodore oder einem damit verbundenen Unternehmen in Zusammenhang. Commodore ist Inhaber des Warenzeichens Amiga.

Vorstand: Carl-Franz von Quadt (Vors.), Lutz Glandt, Dr. Rainer Doll, Dieter Streit
Verlagsleitung: Wolfram Höfler
Operation Manager: Michael Koeppe
Direktor Zeitschriften: Michael M. Pauly
Anschrift des Verlages: Markt & Technik Verlag Aktiengesellschaft, Hans-Pinsel-Straße 2, 8013 Haar bei München

Suchen von Zeichenfolgen in Assembler

Gesucht – gefunden

Bei der Suche nach Zeichenfolgen kann man viel Zeit sparen, wenn man die richtigen Methoden kennt. Hier wollen wir Ihnen drei der schnellsten Algorithmen an Beispielen näherbringen.

von Sebastian Wedeniwski

Wir wollen im folgenden drei Suchalgorithmen betrachten, die eine vorgegebene Zeichenfolge (Muster) im vorgegebenen Speicherbereich suchen. Es sind dies: Brutesearch, KMPSearch und Mischarsearch. Zunächst eine kurze Beschreibung der einzelnen Algorithmen:

Brutesearch: Dieser Algorithmus tritt in vielen Anwendungen auf und ist gut an die Architekturmerkmale der meisten Computersysteme angepaßt, so daß eine optimale Variante einen »Standard« liefert. Bei dieser Suchmethode wird jede mögliche Position im Speicherbereich, an der das Muster passen könnte, überprüft, ob es tatsächlich paßt. Nachteil ist, daß die Suche für manche Muster langsam sein kann, z.B. dann, wenn der Text binär (aus zwei Zeichen gebildet) ist, wie das für Anwendungen bei der Bildverarbeitung der Fall sein kann.

Auf den Algorithmus kommt´s an

KMPSearch: Die Idee, die dem von Knuth, Morris und Pratt entdeckten Algorithmus zugrunde liegt, ist folgender: Wenn eine Nichtübereinstimmung festgestellt wird, besteht der »Fehlstart« aus Zeichen, die bereits bekannt sind (da sie sich im Muster befindet). Um ein einfaches Beispiel hierfür zu betrachten, sei angenommen, daß das erste Zeichen im Muster nicht noch einmal im Muster erscheint (z.B. 100 000). Erfolgt ein Fehlstart an einer gewissen Position im Speicherbereich, wissen wir aufgrund der Tatsache, daß bei den vorhergehenden Zeichen Übereinstimmung vorlag, daß der Speicherzeiger nicht »zurückgesetzt« werden braucht, da keines der vorangehenden Zeichen im Speicher mit dem ersten Zeichen im Muster übereinstimmen kann. Das Auftreten eines solchen Musters ist nicht besonders wahrscheinlich und die Implementation ist recht komplex. Somit ist diese Suchmethode praktisch recht langsam, und nur in den seltensten Fällen effizient. Darüber hinaus benötigt dieser Algorithmus noch eine Tabelle mit der Länge des Musters, um die Sprünge zu gewähren.

Mischarsearch: Dies ist der schnellste Algorithmus (von Boyer-Moore) von den erwähnten Suchmethoden, bei dem bei der Prüfung auf Übereinstimmung mit dem Muster von rechts nach links vorgegangen wird, und wenn sie nicht übereinstimmen, kann sogar festgestellt werden, daß das Zeichen nirgends im Muster auftritt, so daß das Muster sofort ganz an den Zeichen vorbeigeschoben werden kann. Es wird eine Tabelle, mit der Länge 256 (alle Zeichen) Byte, für die erforderlichen Sprünge benötigt. Das Muster darf bei dieser Implementation nicht länger als 255 Zeichen sein.

Der Einsatz der implementierten Algorithmen ist ganz einfach. Die Anfangsadresse des Speicherbereichs, in dem gesucht wird, muß im Adreßregister A1 stehen und die Endadresse in A3. Die Anfangsadresse des Musters muß im Adreßregister A2 stehen und die Endadresse in A4. Anschließend kann eine Suchroutine aufgerufen werden. Das Ergebnis steht dann im Adreßregister A0. Bei einer erfolglosen Suche enthält A0 die Endadresse A3.

Als Test haben wir die drei Suchalgorithmen mit dem Suchbefehl von AMIGA ACTION REPLAY II mit einem beliebigen Muster aus fünf Zeichen verglichen. Der Vergleich lieferte folgendes Ergebnis: Brutesearch war vier-, KMPSearch zwei- und Mischarsearch 15mal schneller als der Such-Befehl. Können Sie´s noch schneller? *ub*

Programmname: Suchen.asm
Assembler: Devpac

```
  1:  lea a,a0      ; Anfangsadresse vom Speicher
  2:  lea Such,a1   ; Anfangsadresse vom Muster
  3:  lea aend,a2   ; Endadresse vom Speicher
  4:  lea Suchend,a3  ; Endadresse vom Muster
  5:  ;bsr Brutesearch
  6:  ;bsr Kmpsearch
  7:  bsr Mischarsearch  ; Ergebnis in a0
  8:  rts
  9: Brutesearch:
 10:  move.l a1,a4
 11: BLoop: move.b (a4)+,d0
 12:  cmp.b (a0)+,d0
 13:  beq.s BLoop2
 14:  sub.l a1,a4
 15:  sub.l a4,a0
 16:  addq.l #1,a0
 17:  move.l a1,a4
 18: BLoop2: cmp.l a2,a0
 19:  bge.s BEnde
 20:  cmp.l a3,a4
 21:  blt.s BLoop
 22:  sub.l a1,a3
 23:  sub.l a3,a0
 24: BEnde: rts
 25: Kmpsearch:
 26:  moveq #0,d0
 27:  move.l d0,d1
 28:  subq.b #1,d1
 29:  lea skip(pc),a4
 30:  lea skip+1(pc),a5
 31:  move.b d1,(a4)
 32:  sub.l a1,a3
 33:  subq #1,a3
 34:  move a3,d3
 35: KLoop4: tst.b d1   ; installieren
 36:  blt.s KLoop
 37:  move.b (a1,d0.w),d2
 38:  cmp.b (a1,d1.w),d2
 39:  beq.s KLoop
 40:  move.b (a4,d1.w),d1
 41:  bra.s KLoop4
 42: KLoop: addq #1,d0
 43:  addq.b #1,d1
 44:  move.b d1,(a5)+
 45:  cmp.b d0,d3
 46:  bgt.s KLoop4
 47:  moveq #0,d1   ; Suchalgorithmus
 48: KLoop8: tst.b d1
 49:  blt.s KLoop5
 50:  move.b (a0),d2
 51:  cmp.b (a1,d1.w),d2
 52:  bne.s KLoop6
 53: KLoop5: addq.l #1,a0
 54:  addq.b #1,d1
 55:  cmp.l a2,a0
 56:  ble.s KLoop7
 57:  rts
 58: KLoop6: move.b (a4,d1.w),d1
 59: KLoop7: cmp.b d3,d1
 60:  ble.s KLoop8
 61:  sub.l d3,a0
 62:  subq.l #1,a0
 63:  rts
 64: Mischarsearch:
 65:  move.l a3,d0
 66:  sub.l a1,d0
 67:  lea (a0,d0.w),a0
 68:  lea skip(pc),a4
 69:  move.l a4,a5
 70:  move.b d0,(a4)+  ; installieren
 71:  move.b d0,(a4)+
 72:  move.b d0,(a4)+
 73:  move.b d0,(a4)+
 74:  move.l (a5),d0
 75:  moveq #62,d7
 76: MLoop: move.l d0,(a4)+
 77:  dbf d7,MLoop
 78:  move.l a1,a4
 79:  moveq #0,d7
 80: MLoop2: move.b (a4)+,d7
 81:  subq #1,d0
 82:  move.b d0,(a5,d7.w)
 83:  bne.s MLoop2
 84:  move.l a3,a4   ; Suchalgorithmus
 85: MLoop4: move.b -(a0),d7
 86:  cmp.b -(a4),d7
 87:  bne.s MLoop3
 88:  cmp.l a4,a1
 89:  beq.s MEnde
 90:  cmp.l a2,a0
 91:  ble.s MLoop4
 92: MEnde: rts
 93: MLoop3: move.b (a5,d7.w),d7
 94:  lea 1(a0,d7.w),a0
 95:  move.l a3,a4
 96:  cmp.l a2,a0
 97:  blt.s MLoop4
 98:  move.l a2,a0
 99:  rts
100: skip: dcb.b 256,0
101: Such: dc.b "Suchen"
102: Suchend:
103: a: dc.b "Speicherbereich als Beispiel zum Su
     chen von Zeichenketten"
104: aend:        ;                    © 1993 M&T
```

»Suchen.asm«: Drei Suchalgorithmen in einem Programm zum Ausprobieren

Mathematische Knobeleien

Denksport

Wenn Sie sich gern mit Zahlenrätseln etc. beschäftigen, haben wir was für Sie: Lösungen von Lesern zu zwei der interessantesten Knobelaufgaben aus der Knobelecke des AMIGA-Magazins.

von Ulrich Brieden

Wer programmiert, tüftelt auch gern. Im AMIGA-Magazin haben wir deshalb speziell für Tüftler die Knobelecke eingerichtet. Jeden Monat finden Sie dort eine interessante Aufgabe aus dem Bereich der Mathematik etc., die es mit dem Computer möglichst elegant und schnell zu lösen gilt. Hier nun zwei Lösungsvorschläge zu Aufgaben des Jahres 1992: Polyominos und Stellensuche (ab Seite 64).

Zu zahlreichen Aufgaben aus der Vergangenheit liegen uns noch weitere interessante Lösungen vor (Damenproblem, Symbolrätsel lösen etc.), die wir auch gerne in diesem Sonderheft abgedruckt hätten; doch wie immer reicht der Platz nicht. Außerdem wollen wir zuerst Ihre Meinung wissen: Wünschen Sie mehr Lösungen zu Knobelaufgaben im AMIGA-Magazin oder im Sonderheft Programmieren? Schreiben Sie uns, damit wir Ihre Wünsche in den nächsten Ausgaben berücksichtigen können.

Polyominos

In Ausgabe 6/92 des AMIGA-Magazins wurde ein Programm gesucht, das Polyominos berechnet. Thomas Eckloff fand eine Lösung in C.
Die Aufgabe in Kurzform:

Ein Polyomino (dt. »Vielzteller«) setzt sich aus miteinander zusammenhängenden quadratischen Zellen zusammen. Es gibt insgesamt fünf verschiedene Tetraminos und zwölf verschiedene Pentominos. Dabei heißen zwei Steine verschieden, wenn sie sich nicht durch Verschiebung, Drehung oder Spiegelung ineinander überführen lassen. Die Tabelle »Verschiedene Polyominos« gibt die Anzahl der verschiedenen Möglichkeiten für die ersten Polyominos mit bis zu sechs Zellen wieder.
Und nun zur Aufgabe! Versuchen Sie, mit Hilfe ihres Amiga die Tabelle zu vervollständigen.

von Thomas Eckloff

Falls man versucht, zur Lösung der Aufgabe die brute-force-Methode anzuwenden, wird man schnell merken, daß die Rechenzeit mit der Zahl der zu berechnenden Teile des Poliominos extrem ansteigt. Schon ab sechs bis sieben Steinen ist nicht absehbar, wann das Programm fertig sein wird. Die in der Aufgabe geforderten 12 Steine zu berechnen, ist so unmöglich.

Es gilt also, Ansätze zu finden, das langwierige Durchsuchen der vorhanden Poliominos zu verkürzen. Die erste deutlich Verkürzung der Rechenzeit erreicht man, wenn man alle Polyominos immer quer speichert, d.h. die breite Seite horizontal.

Dadurch benötigt man für alle nicht rechteckigen Bilder nur vier Darstellungen. Außerdem muß man nur Bilder vergleichen, die dieselbe Höhe und Breite haben; die Rechenzeit reduziert sich dadurch deutlich.

Als nächsten Schritt kann man für den Einstieg in die Liste der Bilder eine Indextabelle erstellen, über die man Zugriff auf Höhe und Breite der Teile hat. Ein Programm muß dann nicht mehr die ganze Tabelle durcharbeiten, sondern kann über die Indizes an die zu vergleichenden Werte kommen. Die Rechenzeit verkürzt sich hierdurch erneut um ein Vielfaches.

Bei abnehmender Rechenzeit erlangt nun ein zweites Problem Bedeutung: Der Speicherplatz wird knapp.

Nehmen wir z.B. die Darstellung »ein Stein entspricht einem Byte«. Für ein 12er Polyomino benötigt man eine 12 * 12-Matrix. Bei ca. 70 000 möglichen Poliominos ergibt das einen Speicherbedarf von 12 12 * 70 000 Byte. Das erfordert nur für die Bilder ca. 10 MByte Speicherbedarf . Ganz abgesehen von der restlichen Informationen, die zu jedem Bild benötigt werden.

Auch mit der Tatsache, daß ein 12er-Polyomino maximal sechs Zeilen hoch sein kann, da es immer quer gespeichert wird, kann man zwar noch 5 MByte sparen, benötigt aber immer noch mehr Speicher als die meisten Amiga-Besitzer haben.

Bei der bitweisen Darstellung benötigt man für eine Zeile maximal 12 Bits. Es bietet sich eine Darstellung in einem Wort (16 Bit) an. Bei sechs Zeilen Höhe ergibt sich ein Speicherbedarf pro Bildinformation von 12 Byte. Die Folge wäre ein Speicherbedarf von 12 * 70 000 = 840 000 Byte für alle Kombinationen. Zu jedem Bild werden noch einmal 4 Byte für die Prüfsumme (Koordinaten + Hashwert) und vier Byte für den Zeiger auf den nächsten Eintrag benötigt, also nochmals 70 000 * 8 = 560 000 Byte.

Nun hat das Amiga-Betriebssystem die Eigenschaft, sich den Speicher immer in Vielfachen von acht Byte zu reservieren. Es würden also nicht nur 12+4+4 = 20 Byte sondern auf das nächste Vielfache von acht aufgerundet, also 24 Byte, reserviert werden. Der Speicherbedarf für eine Liste mit 70 000 Einträgen betrüge also 24 * 70 000 = 1,68 MByte.

VERSCHIEDENE POLYOMINOS (n-ZELLER)

n =	1	2	3	4	5	6	7	8	9	10	11	12
Anzahl =	1	1	2	5	12	35	?	?	?	?	?	?

Ein weiterer Schritt ist es, die Indextabelle zu vergrößern: Über einen Hash-Code kann man Eigenschaften eines Bilds in der Tabelle integrieren, die einen Vergleich erleichtert.

Eine weitere zeitaufwendige Sache ist das ständige Spiegeln und Drehen der Bilder. Falls man die Teile richtig speichert, geht das Ganze relativ schnell. Das vorgestellte Programm »Polyominos.c« speichert alle Polyominios in einer eindeutigen Weise:

Vor dem Durchsuchen prüft es alle acht Perspektiven und merkt sich die höchstwertige. Das ist die Position, die den höchsten Wert darstellt, wenn man sich das Polyomino als Bitmuster vorstellt. Von links nach rechts und dann von oben nach unten betrachtet. Durch dieses Verfahren wird jedes neue Bild automatisch höchstwertig und so gespeichert. Jedes neue Bild muß also nur höchstwertig erklärt zu werden und kann ohne jede weitere Drehung etc, in die Liste eingetragen werden. Beispiel ein Polyomino mit vier Steinen:

```
#   100  #   001  ###  111  ###  111
### 111  ###  111  #    100    #  001
```

Es gibt nur vier Möglichkeiten, weil das Bild immer quer gespeichert wird. In diesem Fall würde das dritte Bild mit der Bit-Kombination 111 100 gespeichert.

Eine bessere Möglichkeit der Bilddarstellung bietet der folgende – zunächst unsinnig erscheindende – Vorschlag, zu jedem Stein zusätzlich seine Koordinaten zu speichern. 12 Steine mit je zwei Koordinaten wären indiskutabel. Bei näherer Betrachtung erkennt man jedoch, daß man mit einem Halbbyte (4 Bit) eine Ordinate darstellen kann. (Werte 0 bis 15). Es fällt auf, daß man für jeden Steine nur einen Wert sichern muß: die Position des Steins in einer Zeile. Beispiel:

```
###.##
.####.
```

Dieses Bild hätte die interne Darstellung zur Folge:

```
1,2,3,5,6,2,3,4,5
```

Ist in so einer Zahlenreihe ein Wert kleiner oder gleich dem vorherigen, hat ein Zeilenwechsel stattgefunden. Ist er größer als der Vorherige, ist der Stein noch in derselben Zeile. Das ist eine Eigenschaft der Polyominios, da immer eine Verbindung zur oberen

Zeile bestehen muß. Um 12 Steine Steine zu speichern, werden nun nur noch 12 Halbbyte also 6 Byte benötigt. Diese 6 Byte zusammen mit den 8 Byte für Checksumme und Zeiger ergeben 14 Byte. Das System belegt allerdings 16 Byte, d.h. wir vergeuden 2 Byte.

Was soll´s? Insgesammt braucht man nur 16 * 70 000 = 1,12 MByte. Das sollte für die meisten Amiga-Besitzer machbar sein.

Da eine Liste von z.B. 10er_Polyominos aus einer Liste von 9er Polyominos aufgebaut wird, sind immer zwei Listen im Speicher. Die alte Liste ist jedoch immer deutlich kleiner als die folgende. Jeder Amiga mit 2 MByte sollte in der Lage sein, ein 12er Poliomnos zu erzeugen. Die echte Anzahl der 12er-Polyominos stellt sich mit 63 600 recht nahan den geschätzten 70 000.

Um das Ganze noch zu vereinfachen, kann man nun folgendes machen. 63 600 Speicheranforderungen belasten das System, zumal man ebensoviele Bereiche am Ende wieder freigeben muß. Es bietet sich also zusätzlich an, eine eigene Speicherverwaltung einzusetzen, die z.B. wie im Beispielprogramm nur alle 100 Bilder Speicher reserviert, der 100 Bilder aufnimmt. Hierdurch verdoppelt sich die Rechengeschwindigkeit im Vergleich zur ersten Methode.

Um alle 63600 Kombinationen mit 12 Steinen zu berechnen, benötigt ein Amiga mit A2630 Turbokarte (25 MHz) etwa 8 Minuten. Ein normaler Amiga rund eine Stunde.

POLYOMINOS

n	7	8	9	10	11	12	13	14
Zahl	108	369	1285	4655	17073	63600	238591	901971

Das Programm ist in C-Standard verfaßt. und enthält keinen Amiga-spezifischen Aufruf. Es läuft genauso unter MS-DOS mit Turbo C und unter Unix auf einer Sun-Workstation (SunOS). Der Autor schaffte mit der Amiga-Version die Berechnung von 13er-Polyominos (238 591). Die Sun-Workstation berechnete 14er Polyominos in knapp 35 Minuten (901 971). Die Tabelle zeigt die gesuchten Werte.

Mit dem Programm sind maximal 14er Polyominos berechenbar. Der Aufwand für die Berechnung größerer Polyominos ist gering und liegt hauptsächlich in der Änderung vom Halbbyte auf Byte bei der internen Bilddarstellung. Das Programm benötigt zwei Parameter: Der erste gibt die Anzahl der gewünschten Steine an und der zweite die Breite der Ausgabe. Der zweite Parameter ist optional und beträgt standardmäßig 79. *ub*

```
  1: struct pol
  2:  {
  3:    unsigned long x,y;
  4:    unsigned long cs;
  5:    unsigned char *image;
  6:    struct pol *np;
  7:  };
  8: struct poln
  9:  { unsigned long cs;
 10:    unsigned char image[8];
 11:    struct poln *np;
 12:   };
 13: struct meml
 14:  { struct poln sp[100];
 15:    struct meml *nm;
 16:  };
 17: struct pol *pold1,*pold2,*pold3,*pold4;
 18: struct poln *palt,*pneu,*lpoff,*poldn,*p1,
     *poff;
 19: unsigned long poll,polnl,poli,polineu,poli
     n,polipix,polix,poliy,offx,offy,tiefe,h,b,
     dis=80L,anz=0,anzx,anzd,ox,oy,adr,pich,pic
     b,pico,picn,picz=500,ent=8192,blks=100,mem
     p,interv;
 20: void *malloc();
 21: struct meml *malloc_own();
 22: unsigned char *imagex,imageh[16],*bild;
 23: struct poln **Entry;
 24: struct meml *memp1,*memp2,*memakt;
 25: main(argc,argv)
 26: int argc;
 27: unsigned char *argv[];
 28: {struct poln *p1;
 29:  if(argc < 2) /* Wenn kein Parameter angeg
     eben wurde, standardmäßig 5 Steine */
 30:   polin=5;
 31:  else /* sonst Anzahl nach >polin< */
 32:   sscanf(argv[1],"%ld",&polin);
 33:  if(argc > 2) /* Zeichen je Zeile für Ausg
     abe - Standard 80, bei 0 keine A. */
 34:   sscanf(argv[2],"%ld",&dis);
 35:  Entry=(struct poln **) malloc((int)(ent*4
     )); /* Speicher für Hashtabelle reservieren */
 36:  if(!Entry)
 37:    printf("\nFehler bei malloc\n");
 38:  bild=malloc((int)(picz*16));   /* Speiche
     r für Ausgabebereich reservieren (500 x 16
      Zeichen) */
 39:  if(!bild)
 40:    printf("\nFehler bei malloc\n");
 41:  Init();  /* Diverse Initialisierungen  */
 42:  memp1=(struct meml *)-1L; /* Pointer für
     eigene Speicherverwaltung initialisieren */
 43:  memp2=(struct meml *)-1L;
 44:  pold1=(struct pol *) malloc((int)poll);
 45:  /* Initialisieren von vier Zwischenspeich
     ern für die interne Darstellung der Polyom
     inos mit 16 x 16 Punkten. Die Größe wurde */
 46:  /* aus dem Grunde statisch gewählt, da so
      bei der Adressberechnung statt einer Mult
     iplikation ein Linksshift um vier Bits mög
     lich ist. */
 47:  pold1->x=polineu;
 48:  pold1->y=polineu;
 49:  pold1->image=malloc((int)256);
 50:  pold1->np=(struct pol *)-1L;
 51:  pold2=(struct pol *)malloc((int)256);
 52:  pold2->x=polineu;
 53:  pold2->y=polineu;
 54:  pold2->image=malloc((int)256);
 55:  pold2->np=(struct pol *)-1L;
 56:  /* In diesen Zwischenspeichern wird jeder
      gesetzte Stein in */
 57:  /* einem Byte abgelegt, was bei späteren
      Manipulationen (Drehung, Spiegelung, etc.)
      schneller ist als bei bit orientierter Sp
      eicherung. */
 58:  pold3=(struct pol *)malloc((int)poll);
 59:  pold3->x=polineu;
 60:  pold3->y=polineu;
 61:  pold3->image=malloc((int)256);
 62:  pold3->np=(struct pol *)-1L;
 63:  pold4=(struct pol *)malloc((int)poll); /*
      Struktur pol  -> byteweise  */
 64:  pold4->x=polineu;    /* Struktur poln ->
     bitweise  */
 65:  pold4->y=polineu;
 66:  pold4->image=malloc((int)256);
 67:  pold4->np=(struct pol *)-1L;
 68:  poldn=(struct poln *)malloc((int)polnl);
     /* Initialisieren einer poln-Struktur  */
 69:  poldn->np=(struct poln *)-1L;
 70:  Poliomino(); /* Aufruf Hauptprogramm */
 71:  anzd=picn=pico=pich=picb=0;
 72:  ClearImage(); /* Löschen des Ausgabebereichs*/
 73:  while(pneu != (struct poln *)-1L)  /* Aba
     rbeiten der Liste der erzeugten Polyominos
      und  Ausgabe + Zählen  */
 74:  {  if(dis > 0L)
 75:      PicOut(pneu,polineu);
 76:     anzd++;
 77:     p1=pneu;
 78:     pneu=pneu->np;
 79:  }
 80:  memp1=memp2;
 81:  free_own(); /* Freigeben des Speichers  * /
 82:  if(dis > 0L) /* Ausgeben des letzten Ausg
     abebereichs */
 83:     Out_Line();
 84:  free(poldn,polnl);  /* Freigabe des Speic
     her der Zwischenstrukturen */
 85:  free(pold4->image,256);
 86:  free(pold4,poll);
 87:  free(pold3->image,256);
 88:  free(pold3,poll);
 89:  free(pold2->image,256);
 90:  free(pold2,poll);
 91:  free(pold1->image,256);
 92:  free(pold1,poll);
 93:  free(bild,picz*16L); /* Freigabe des Ausg
     abebereichs  */
 94:  free(Entry,ent*4L);  /* Freigabe der Hash
     tabelle  */
 95:  printf("\n%10ld Poliominos ermittelt\n",anzd);
 96: }
 97: Poliomino()
 98: {for(poli=1;poli<polin;poli++)
          /* Hauptschleife   */
 99:      Poli_1();
100: }
101: Init()
102: {memp=blks+1; /* Initialisierung der verke
     tteten Listen für die Polyominos und die S
      peicherverwaltung */
103:  memp2=(struct meml *)-1L;
104:  memakt=(struct meml *)-1L;
105:  poll=sizeof(struct pol); /* Dabei wird da
     s Ausgangspolyomino (ein einziger Stein) "
     per Hand" erzeugt */
106:  polnl=sizeof(struct poln);
107:  poli=1;
108:  palt=(struct poln *)-1L;
109:  pneu=(struct poln *) malloc_own();
110:  pneu->cs=0x00000011;
111:  pneu->image[0]=0x10;
112:  pneu->np=(struct poln *)-1L;
113:  tiefe=1;
114: }
115: Poli_1()
116: {register unsigned long i;
117:  /* In dieser Routine wird die jeweils zul
     etzt erzeugte Liste mit Polyominos abgearb
     eitet und die neue Liste aufgebaut */
118:  struct poln *p1;
119:  polineu=poli+2;
120:  palt=pneu;
121:  anzx=0;
122:  pneu=(struct poln *)-1L;
123:  poff=(struct poln *)-1L;
124:  memp1=memp2;
125:  memakt=(struct meml *)-1L;
126:  memp=blks+1;
127:  for(i=0;i<ent;i++) /*Löschen der Hashtabelle*/
128:     Entry[i]=0L;
129:  anzd=0;
130:  interv=0;
131:  while(palt != (struct poln *)-1L)
132:  /* Abarbeiten der alten Liste  */
133:  { /* und Aufruf eines Unterprogramms, das
      aus jedem alten Bild versucht, mehrere ne
     ue zu erstellen */
134:   Poli_2();
135:     p1=palt;
136:     palt=palt->np;
137:  }
138:  free_own(); /* Freigabe des */
139:    /* Speichers der alten Struktur  */
140:  printf("%10ld%10ld\n",poli+1,anzd);  /* A
     usgabe eines Zwischenstandes */
141: }
142: Poli_2()
143: {register unsigned long i,j;
144:  register unsigned char *p1,*p2;
145:  j=256;
146:  p1=pold1->image;
147:  p2=pold2->image;
148:  for(i=0;i<j;i++) /* Löschen der Bildspeicher*/
149:  {       /* von pold1 und pold2   */
150:     *p1++=0;
151:     *p2++=0;
152:  }
153:  Unpack(palt); /* Entpacken des nächsten  */
154:       /* Eintrags in der Liste  */
155:  p1=pold1->image;
156:  p2=pold2->image;
157:  for(i=1;i<=pold1->y;i++)
158:  /* Das entpackte Bild wird Zeile für Zeil
     e, Zeichen für Zeichen abgearbeitet. Bei e
```

```
    inem nicht leeren Feld wird versucht, oben
    , unten, rechts und links ein Stein anzufügen*/
159:     for(j=1;j<=poldl->x;j++)
160:        if(*(p1+(i<<4)+j) != 0)
161:        {  Poli_5(p1,p2,i,j-1);
162:           Poli_5(p1,p2,i-1,j);
163:           Poli_5(p1,p2,i,j+1);
164:           Poli_5(p1,p2,i+1,j);
165:        }
166: }
167: Poli_5(p1,p2,x,y)
168: register unsigned char *p1,*p2;
169: register unsigned long x,y;
170: {register unsigned long x1;
171:  x1=(x<<4)+y;
172:  if(*(p2+x1) == 1) /* Wenn dort schon ein
    Stein war */
173:     return(0);      /* ->Rücksprung     */
174:  *(p1+x1)=1;      /* Eintragen des Steins
    */
175:  *(p2+x1)=1;      /* Position als besetzt k
    ennzeichnen */
176:  ox=1; /* Feststellen, ob sich die Höhe */
177:  oy=1; /* oder die Breite geändert hat  */
178:  b=poldl->x; /* oder ob in den oberen oder */
179:  h=poldl->y; /* linken Rahmen geschrieben
    wurde */
180:  if(x == 0L)
181:  {  ox--;
182:     h++;
183:  }
184:  if(x > h)
185:     h++;
186:  if(y == 0L)
187:  {  oy--;
188:     b++;
189:  }
190:  if(y > b)
191:     b++;
192:  Poli_7(x,y);
193:  *(p1+x1)=0;      /* Herstellen der Origina
    lbildes */
194: }
195: Poli_7(x,y)
196: unsigned long x,y;
197: {register unsigned char *p1,*p2;
198:  /* Das neu entstandene Bild wird bündig n
    ach oben links ausgerichtet. */
199:  register unsigned long i,j,l;
200:  p1=pold1->image+(ox<<4)+oy;
201:  p2=pold3->image;
202:  pold3->x=b;
203:  pold3->y=h;
204:  for(i=0;i<h;i++)
205:  { for(j=0;j<b;j++)
206:        *p2++=*p1++;
207:     p1+=(16-b);
208:     p2+=(16-b);
209:  }
210:  if(h > b)  /* Wenn die Höhe des Bildes  */
211:     Rot_90(); /* größer ist als die Breite
    , wird es um 90 Grad gedreht  */
212:  Poli_9();
213: }
214: Rot_90()
215: {register unsigned char *p1,*p2,zeichen; /
    * In diesem Unterprogramm */
216:  register unsigned long i,j,i1,i2,x; /* f
    indet die Drehung durch Spiegelung an der
    Diagonalen  statt */
217:  p1=pold3->image;
218:  x=pold3->x;
219:  pold3->x=pold3->y;
220:  pold3->y=x;
221:  p1=pold3->image;
222:  p2=pold3->image;
223:  for(i=0;i<pold3->x;i++)
224:  {  i1=i<<4;
225:     i2=i;
226:     for(j=0;j<i;j++)
227:     {  zeichen=*(p1+i1);
228:        *(p1+i1)=*(p2+i2);
229:        *(p2+i2)=zeichen;
230:        i1++;
231:        i2+=16;
232:     }
233:  }
234: }
235: Poli_9()
236: {register struct poln *p1,*p2;
237:  register unsigned long i,k,offset;
238:  p1=pneu;
239:  p2=(struct poln *)-1L;
240:  Normalize(); /* Beschreibung siehe Routine */
241:  Copy_Pol(pold3,pold4);  /* Es wird das hö
    chstwertige Bild übernommen */
242:  Checksum();  /* Ermitteln des Hashindexes  */
243:  Pack(pold3);      /* Packen zum Vergleich  */
244:  adr=poldn->cs>>8; /* Hashadresse ermitteln  */
245:  for(i=adr;i>0;i--)
246:  { if(Entry[i] != 0L)  /* Ersten Eintrag <
    = dem gesuchten ermitteln */
247:     {  p1=Entry[i];
248:        break;
249:     }
250:  }
251:  k=0;
252:  while((p1 != (struct poln *)-1L) && (k ==
    0)) /* Durchsuchen der vorhandenen Bilder */
253:  { if((p1->cs>>8) > adr)
254:       break;
255:     if(p1->cs == poldn->cs)
256:     if(strncmp(p1->image,poldn->image,8) == 0)
257:          k=1;
258:       else
259:          k=0;
260:     p2=p1;
261:     p1=p1->np;
262:  }
263:  if(k == 1) /* schon vorhanden -> Rücksprung */
264:     return(0);
265:  Poli_10(p1,p2); /* Aufnehmen des Bildes    */
266: }
267: Poli_10(lp,lp2)
268: struct poln *lp,*lp2;
269: {register struct poln *p1;   /* Einstellen
    in die Kette  */
270:  register unsigned char *p2,*p3;
271:  register unsigned long i,j;
272:  p1=poff;
273:  poff=(struct poln *)malloc_own();
274:  poff->cs=poldn->cs;
275:  strncpy(poff->image,poldn->image,8);
276:  poff->np=lp;
277:  if((lp == (struct poln *)-1L) && (lp2 ==
    (struct poln *)-1L))
278:     pneu=poff;
279:  else
280:     if(lp2 == (struct poln *)-1L)
281:     {  pneu=poff;
282:        poff->np=lp;
283:     }
284:     else
285:        lp2->np=poff;
286:  if(Entry[adr] == 0L)     /* Eintrag in die
    Hashtabelle */
287:     Entry[adr] = poff;
288:  anzd++;
289:  interv++;
290:  if(interv == 100)     /* Alle 100 neuen Bi
    lder auf dem Bildschirm zeigen */
291:  {  printf("%10ld%10ld\r",poli+1,anzd);
292:     interv=0;
293:  }
294: }
295: Mirror_H()  /* Die Routine spiegelt ein  */
296: {      /* Bild horizontal     */
297:  struct pol *p1;
298:  register unsigned char *p3,*p4,zeichen,*p2;
299:  register unsigned long i,j,k,l;
300:  l=pold3->x/2;
301:  p1=pold3;
302:  p2=p1->image;
303:  for(i=0;i<pold3->y;i++)
304:  {  k=pold3->x-1;
305:     p3=p2+pold3->x;
306:     p4=p2;
307:     for(j=0;j<l;j++)
308:     {  zeichen=*p4;
309:        *(p4++)=*(--p3);
310:        *p3=zeichen;
311:     }
312:     p2+=16;
313:  }
314: }
315: Mirror_V()  /* Die Routine spiegelt ein  */
316: {     /* Bild vertikal    */
317:  struct pol *p1;
318:  register unsigned char *p3,*p4,zeichen,*p2;
319:  register unsigned long i,j,k,l;
320:  p1=pold3;
321:  p2=p1->image;
322:  k=0;
323:  l=(pold3->y-1)<<4;
324:  for(i=0;i<pold3->y/2;i++)
325:  {  p3=p2+k;
326:     p4=p2+l;
327:     for(j=0;j<pold3->x;j++)
328:     {  zeichen=*p3;
329:        *(p3++)=*p4;
330:        *(p4++)=zeichen;
331:     }
332:     k+=16;
333:     l-=16;
334:  }
335: }
336: Copy_Pol(p1,p2) /* Die Routine kopiert eine */
337: struct pol *p1,*p2; /* komplette Struktur   */
338: {register unsigned char *p3,*p4;
339:  register unsigned long i,j,k;
340:  p3=p1->image;
341:  p4=p2->image;
342:  k=pold3->x;
343:  for(i=0;i<pold3->y;i++)
344:  {for(j=0;j<k;j++)
345:        *p3++=*p4++;
346:     p3+=16-k;
347:     p4+=16-k;
348:  }
349: }
350: Cmp_Pol(p1,p2) /* Die Routine vergleicht  */
351: struct pol *p1,*p2;     /* zwei Bilder  */
352: {register unsigned char *p3,*p4;
353:  register unsigned long i,j,k;
354:  p3=p1->image;
355:  p4=p2->image;
356:  for(i=0;i<pold3->y;i++)
357:     for(j=0;j<pold3->x;j++)
358:     {  k=*(p3+(i<<4)+j);
359:        k=k-*(p4+(i<<4)+j);
360:        if(k != 0)
361:          return(k);
362:     }
363:  return(0);
364: }
365: Normalize()  /* Hier wird das  */
366: {     /* Bild auf alle möglichen  */
367:  Copy_Pol(pold4,pold3); /* Darstellungen g
    edreht und gespiegelt. */
368:  Mirror_H();  /* Dabei wird das Bild mit d
    er höchsten Wertigkeit, d. h. */
369:  if(Cmp_Pol(pold3,pold4) > 0)   /* mit dem
    frühesten Auftreten  */
370:     Copy_Pol(pold4,pold3);   /* ausgefüllt
    er Felder (von  */
371:        /* rechts nach links und von   */
372:  Mirror_V(); /* oben nach unten) in pold4  */
373:  if(Cmp_Pol(pold3,pold4) > 0)   /* abgelegt.*/
374:     Copy_Pol(pold4,pold3);
375:  Mirror_H();
376:  if(Cmp_Pol(pold3,pold4) > 0)
377:     Copy_Pol(pold4,pold3);
378:  if(pold3->x != pold3->y)
379:     return(0);
380:  Rot_90();
381:  if(Cmp_Pol(pold3,pold4) > 0)
382:     Copy_Pol(pold4,pold3);
383:  Mirror_H();
384:  if(Cmp_Pol(pold3,pold4) > 0)
385:     Copy_Pol(pold4,pold3);
386:  Mirror_V();
387:  if(Cmp_Pol(pold3,pold4) > 0)
388:     Copy_Pol(pold4,pold3);
389:  Mirror_H();
390:  if(Cmp_Pol(pold3,pold4) > 0)
391:     Copy_Pol(pold4,pold3);
392:  return(0);
393: }
394: Checksum()      /* Ermitteln der Checksumme  */
395: {register unsigned char *p1;
396:  register unsigned long i,j,k,l;
397:  k=0;
398:  l=0;
399:  p1=pold3->image;
400:  for(i=0;i<pold3->y;i++)
401:     for(j=0;j<pold3->x;j++)
402:        k+=(*(p1+(i<<4)+j)*i<<10+j);
403: }
404: Unpack(p)      /* Entpacken eines Bildes  */
405: struct poln *p;
406: {register unsigned char *p1,*p2,*p3;
407:  register unsigned long i,j,k,ln,la,x,y;
408:  i=p->cs&0xff;
409:  x=i/16;
410:  y=i&0xf;
411:  j=0;
412:  pold1->x=x;
413:  pold1->y=y;
414:  p1=pold1->image;
415:  p2=pold2->image;
416:  imagex=(unsigned char *) &p->image[0];
417:  p3=&imageh[0];
418:  for(i=0;i<8;i++)
419:  { j=imagex[i];
420:     *p3++=(j>>4)&0xf;
421:     *p3++=j&0xf;
422:  }
423:  x=0;
```




```
424:  y=0;
425:  la=99;
426:  for(k=0;k<poli;k++)
427:  {  ln=imageh[k];
428:     if(ln <=la)
429:       y++;
430:     x=ln;
431:     la=ln;
432:     *(p1+(y<<4)+x)=1;
433:     *(p2+(y<<4)+x)=1;
434:  }
435: }
436: Pack(p)      /* Packen eines Bildes    */
437: struct pol *p;
438: {register unsigned long i,j,k,l,m;
439:  register unsigned char *p1,*p2;
440:  p1=p->image;
441:  p2=&imageh[0];
442:  for(i=0;i<16;i++)
443:     *p2++=0;
444:  p2=&imageh[0];
445:  k=0;
446:  for(i=0;i<p->y;i++)
447:  {  l=0;
448:     for(j=0;j<p->x;j++)
449:     {  m=*(p1+(i<<4)+j);
450:        if(m != 0)
451:        { *p2++=j+1;
452:          l+=(1<<j);
453:        }
454:     }
455:     k=k^l;
456:     k*=9;
457:  }
458:  p2=&imageh[0];
459:  imagex=(unsigned char *) &poldn->image[0];
460:  for(i=0;i<8;i++)
461:  {  j=*p2++<<4;
462:     j+=*p2++;
463:     imagex[i]=j;
464:  }
465:  poldn->cs=pold3->x*16+pold3->y;
466:  poldn->cs+=((k&0x1fff)<<8);
467: }
468: PicOut(p,n)      /* Ausgabe der Polyominos  */
469: register struct poln *p;
470: register unsigned long n;
471: {unsigned char *p1,*p2,*p3;
472:  unsigned long i,j,k,ln,la,x,y,h,b;
473:  anz++;
474:  i=p->cs&0xff;
475:  x=i/16;
476:  y=i&0xf;
477:  j=0;
478:  b=x;
479:  h=y;
480:  if(pico + b > dis)
481:     Out_Line();
482:  if(h > pich)
483:     pich=h;
484:  x=0;
485:  y=0;
486:  la=0;
487:  imagex=(unsigned char *) &p->image[0];
488:  p3=&imageh[0];
489:  for(i=0;i<8;i++)
490:  { j=imagex[i];
491:    *p3++=(j>>4)&0xf;
492:    *p3++=j&0xf;
493:  }
494:  x=0;
495:  y=0;
496:  la=0;
497:  k=0;

498:  while(ln=imageh[k])
499:  { k++;
500:     if(ln <=la)
501:       y++;
502:     x=ln;
503:     la=ln;
504:     x--;
505:     *(bild+pico+y*picz+x)='#';
506:  }
507:  pico+=b+1;
508: }
509: Out_Line()
510: {register unsigned long i,j;
511:  for(i=0;i<pich;i++)
512:  {  *(bild+i*picz+pico)=0;
513:     printf("%s\n",bild+i*picz);
514:  }
515:  printf("-----\n");
516:  pich=0;
517:  picb=0;
518:  pico=0;
519:  ClearImage();
520: }
521: ClearImage()
522: {register unsigned char *p1;
523:  register unsigned long i,j;
524:  p1=bild;
525:  for(i=0;i<picz*16;i++)
526:     *p1++=' ';
527: }
528: struct meml *malloc_own()   /* eigene Spei
     cherverwaltung dadurch muß nur alle 100 */
529: {
530:  register struct meml *p1;   /* Bilder ein
     malloc durchgeführt werden.   */
531:  memp++;
532:  if(memp >= blks)
533:  {  memp=0;
534:     p1=memakt;
535:     memakt=(struct meml *)malloc(sizeof (s
    truct meml));
536:     if(p1 == (struct meml *)-1L)
537:        memp2=memakt;
538:     else
539:        p1->nm=memakt;
540:     memakt->nm=(struct mel *)-1L;
541:  }
542:  return((struct meml *)&memakt->sp[memp]);
543: }
544: free_own()     /* Freigabe der Speicherliste*/
545: {register struct meml *p1,*p2;
546:  p1=memp1;
547:  while(p1 != (struct meml *)-1L)
548:  {  p2=p1;
549:     p1=p1->nm;
550:     free(p2,sizeof(struct meml));
551:  }
552: }
     © 1993 M&T
```

»Polyominios.c«: Das Programm berechnet Polyominos mit bis zu 14 Teilen

Amiga geht auf Stellensuche

In der Knobelecke 9/1992 des AMIGA-Magazins stellten wir die Aufgabe, ein Programm zu schreiben, um Pi auf möglichst viele Stellen zu berechen. Heiko Hirschmüller aus schaffte mit einem Assembler-Programm die Millionengrenze.

von Heiko Hirschmüller

Das Programm »Pi.c« wird vom CLI mit
`Pi n      (bzw.    Pi n >Datei)`
gestartet, wobei n die Anzahl der gewünschten Nachkommastellen ist. (Bsp. »Pi 2000«, berechnet Pi mit 2000 Nachkommastellen und gibt das Ergebnis auf dem Bildschirm aus.) Die Zahl n sollte zwischen 12 und 1 000 000 liegen und wird intern gerundet, so daß sie ohne Rest durch 4 teilbar ist.
Funktionsweise des Programms:
Das Programm berechnet Pi nach :
`Pi = 48*arctan (1/18) + 32*arctan (1/57) - 20*arctan (1/239)`
wobei der arctan durch die Taylor`sche Reihe
`arctan (x) = x - x^3/3 + x^5/5 - x^7/7 + ...`
angenähert wird.
Nach dem Einlesen der gewünschten Stellenanzahl aus der Kommandozeile und dem Öffnen der »dos.library« wird der benötigte Speicher reserviert. Nun werden nacheinander die drei arctan-Terme berechnet. Dabei wird zunächst der Zähler (z.B. 48 beim 1. Term) in den Speicherbereich des Summanden geschrieben (nach dem 1. Wort beginnen die Nachkommastellen). Danach wird der Summand durch das Argument geteilt (Beim 1. Term 18) und anschließend wird der Summand (der nun das 1. Glied des Taylorpolynoms ist) zur Summe hinzuaddiert (oder Subtra hiert beim 3. Term).
In der folgenden Schleife wird jeweils das folgende Glied der Taylorreihe berechnet (Summand:=Summand/(Argument^2) und Zwischenspeicher:=Summand/i, wobei Argument beim 1. Term 18 ist und i := 3, 5, 7, 9, 11, ...) und abwechselnd zur Summe addiert oder davon abgezogen. Die Schleife wird solange durchlaufen, bis der Summand null ist. Danach folgt die Umrechnung in eine Dezimalzahl und Ausgabe auf dem Bildschirm in ähnlicher Weise wie beim Beispielprogramm (Berechnung von e) AMIGA-Magazin 9/92, Seite 58)
Es bleibt noch zu bemerken, daß das gesammte Programm zur Berechnung von Pi mit mehr als 1 Million Stellen ausgelegt ist. Insbesondere mußte der Autor eine Divisionsart finden, bei welcher der Divisor größer als 65535 ist, sonst wäre eine Berechnung von Pi nur mit max. 82 268 Stellen möglich.
Im zugehörigen Programmteil wird der Divisor zunächst auf einen 16-Bit-Wert vermindert, um so (durch normale Division) daß Ergebnis abzuschätzen. Danach wird das Ergebnis und der Rest noch korregiert (Die Kommentare im Source-Code dürften zum Verständnis ausreichen).
Der Rundungsfehler bei 250000 Stellen dürfte maximal die letzten sechs Stellen betreffen. Zur Berechnung des 1. Terms sind 199 157 Glieder notwendig, bei denen maximal die letzte Stelle um +-1 falsch ist. Bei Addition des maximalen Fehlers ergeben sich so sechs unsichere Stellen. Bei Berücksichtigung der anderen zwei arctan-Terme ist man ebenfalls mit sechs unsicheren Stellen auf jeden Fall auf der sicheren Seite.
Bei der Berechnung von Pi mit 100000 Nachkommastellen stimmt (im Vergleich zu Pi mit 250000 Stellen) sogar noch die letzte Stelle, da intern ein paar Byte mehr reserviert werden, als zur Darstellung der Dezimalzahl nötig. Zum Vergleich sind auf der Diskette zum Heft die Ergebnisse von »Pi 100000« und »Pi 250000« als Text gespeichert.
Ausführungszeit: Die Ausführungszeit wurde auf einem normalen Amiga 500 (7.14 Mhz) gemessen. (vom Start des Programms, bis zur Beendigung der Bildschirmausgabe). Die folgende Tabelle zeigt die Ergebnisse. Versuchen Sie, das Programm doch entsprechend auszubauen und zu verbessern, daß es noch schneller wird. Viel Erfolg. *ub*

Anzahl Nachkommastellen	benötigte Zeit
1000	6 sec
10000	8 min 52 sec
100000	14 Stunden 48 min 49 sec
250000	4 Tage 23 Stunden 44 min 16 sec
1 Million	79 Tage 19 Stunden (geschätzt)

```
1: * Pi, mit max. 1000000 Nachkommastellen   *
2: * Pi=48*arctan (1/18)+32*arctan (1/57)-20*a
   rctan (1/239) *
3: * (Berechnung acrtan mit Taylorpolynom) *
4: * von HEIKO HIRSCHMÜLLER *
5: * Computer  : Amiga 500, Kickstart 2.0 *
6: * Assembler : Devpac V 2.0              *
7: Lenght   equ  4096 * Länge Ausgabepuffer
8: ExecBase equ  4    * Libraryfunktionen
9: AllocMem equ  -198
10: FreeMem  equ  -210
11: OpenLib  equ  -552
12: CloseLib equ  -414
13: Output   equ  -60
14: Write    equ  -48
15: * Anzahl der Nachkommastellen lesen
16:   moveq   #0,d0   * Anzahl der gewünschten
17:   moveq   #0,d1   * Nachkommastellen aus
18:   move.w  #48,d2  * Kommandozeile lesen und
19: NextZiff move.l d0,d3 * in binäre Zahl wand
   eln
20:   lsl.l   #2,d3
21:   add.l   d3,d0
22:   add.l   d0,d0
23:   add.l   d1,d0
24:   move.b  (a0)+,d1
25:   sub.b   d2,d1
26:   blt     Vergl   * list, bis keine
27:   cmp.b   #9,d1   * Ziffern mehr vorhanden
28:   bls     NextZiff
29: Vergl    cmp.l   #12,d0 * Anzahl zwischen
30:   bhi     groesser  * 12 und 1000000
31:   moveq   #12,d0
32: groesser cmp.l   #1000000,d0
33:   bls     kleiner
34:   move.l  #1000000,d0
35: kleiner  and.b   #$fc,d0 * durch 4 teilbar
36:   move.l  d0,DezLen * berechnete Zahl sichern
37: * Länge des benötigten Speichers berechnen
38:   move.l  d0,d1   * Länge in Bytes :=
39:   swap    d1      * (Anz. Dez. Stellen)*4152
40:   mulu    #1038,d1  * /10000+8
41:   swap    d1
42:   mulu    #1038,d0
43:   add.l   d1,d0
44:   moveq   #0,d1
45:   swap    d0
46:   move.w  d0,d1
47:   divu    #10000,d1
48:   swap    d1
49:   move.w  d1,d0
50:   swap    d0
51:   divu    #10000,d0
52:   move.w  d0,d1
53:   lsl.l   #2,d1
54:   add.l   #8,d1
55:   move.l  d1,laenge * und sichern
56: * Dos-Library öffnen, Output-Handle holen
57:   lea     dosname(pc),a1 * Dos-Lib. öffnen
58:   moveq   #0,d0
59:   move.l  ExecBase,a6
60:   jsr     OpenLib(a6)
61:   tst.l   d0
62:   beq     ende
63:   move.l  d0,dosbase
64:   move.l  d0,a6         * Output-Handle
65:   jsr     Output(a6)    * holen
66:   move.l  d0,handle
67: * benötigten Speicher reservieren
68:   moveq   #2,d7           * Speicher für
69:  lea     Summand(pc),a5 * Summanden, Zwischen
70: reserv  move.l  laenge(pc),d0 * Summe reserv.
71:   moveq   #1,d1
72:   swap    d1
73:   move.l  ExecBase,a6
74:   jsr     AllocMem(a6)
75:   move.l  a1,(a5)+
76:   tst.l   d0
77:   beq     abbruch
78:   dbra    d7,reserv
79: * Startwerte des 1. Summanden von arctan
80:   lea     Startwert(pc),a3 * ->Startwerte
81:   moveq   #2,d3         * 3 Terme berechnen
82: Do  move.l  Summand(pc),a5 * -> Summand
83:   move.l  laenge(pc),d5 * Länge in Worten
84:   lsr.l   d5
85:   move.w  (a3)+,(a5) * Multipl.-> Summand
86:   move.w  (a3),d6    * Divisior
87:   move.l  d5,d4      * Summand :=
88:   subq.l  #1,d4      * Summand/Divisor
89:   swap    d4
90:   move.l  a5,a1
91:   moveq   #0,d0
92: Divid1   swap    d4
93: Divid2   move.w  (a1),d0
94:   divu    d6,d0
95:   move.w  d0,(a1)+
96:   dbra    d4,Divid2
97:   swap    d4
98:   dbra    d4,Divid1
99:   move.l  d5,d4        * Summe:=
100:  lsr.l   d4           * Summe +,- Summand
101:  subq.l  #1,d4
102:  swap    d4
103:  move.l  Summe(pc),a0
104:  add.l   d5,a0
105:  add.l   d5,a0
106:  tst.w   2(a3)        * + oder - wird fest-
107:  blt     SubLoop1     * gelegt bei Start-
108: SumLoop1 swap   d4    * werten.
109: SumLoop2 addx.l  -(a1),-(a0)
110:  dbra    d4,SumLoop2
111:  swap    d4
112:  dbra    d4,SumLoop1
113:  bra     LoopEnd
114: SubLoop1 swap    d4
115: SubLoop2 subx.l  -(a1),-(a0)
116:  dbra    d4,SubLoop2
117:  swap    d4
118:  dbra    d4,SubLoop1
119: LoopEnd
120: * Berechnung des folgenden Summanden des
121: * Taylorpolynoms und Addition zur Summe
122:  mulu    (a3)+,d6 * Divisor ist quadratisch
123:  moveq   #3,d7    * Startwert des Zählers i
124:  move.l  Zwischen(pc),a4
125: Taylor   jsr     div(pc)
126:  * Summand:=Summand/Divisor - Zwischen:=Sum
   mand/i
127:  jsr     summiere(pc) * Summe:=Summe +,- Z
   wischen
128:  addq.l  #2,d7     * i:=i+2
129:  tst.l   (a5)      * Wenn die ersten Ziffern
130:  bne     Taylor    * des Summanden null sind,
131:  moveq   #4,d0
132:  adda.l  d0,a5     * dann können diese über-
133:  adda.l  d0,a4     * sprungen werden.
134:  subq.l  #2,d5
135:  bgt     Taylor
136:  adda.l  #2,a3     * Für alle 3 Terme die
137:  dbra    d3,Do     * gleiche Prozedur
138: * Ergebnis in Dez. Zahl wandeln und ausgeben
139:  move.l Summe(pc),a4 * Ziffer in 1. Wort von
140:  move.l  laenge(pc),d7  * Summe und ein
   Komma
141:  lea     Ausgabe(pc),a5 * in Ausgabepuffer
142:  move.w  #$2020,d0      * schreiben.
143:  swap    d0
144:  move.w  (a4),d0
145:  add.w   #48,d0
146:  lsl.w   #8,d0
147:  move.b  #'.',d0
148:  move.l  d0,(a5)+
149:  move.l  #Lenght/4-2,d3 * restl. Länge in d3
150:  move.w  #0,(a4)
151:  add.l   d7,a4          * Ende von Summe -> a4
152:  lsr.l   #1,d7          * Länge von Summe
153:  subq.l  #1,d7          * in Worten
154:  move.l  DezLen(pc),d5 * Anz. der
155:  lsr.l   #2,d5           * Dezimalstellen/4
156: red      move.l  a4,a0  * Startwerte der
157:  move.l  d7,d6          * Reduktionsschleife
158:  swap    d6
159:  move.w  #10000,d4
160:  moveq   #0,d1
161: red1  swap    d6     * Summe mit 10000
162: red2  move.w  -(a0),d0 * multiplizieren
163:  mulu    d4,d0         * und ...
164:  add.l   d1,d0
165:  move.w  d0,(a0)
166:  swap    d0
167:  move.w  d0,d1
168:  dbra    d6,red2
169:  swap    d6
170:  dbra    d6,red1
171:  move.w  d0,(a0)      * ... Übertrag
172:  swap    d0           * ausgeben
173:  jsr     print(pc)
174:  subq.l  #1,d5
175:  bgt     red          * und nochmal
```



```
176:   jsr     Out(pc)  * Rest Ausgabepuffer
177:   move.l  #$2e2e2e0a,Ausgabe
178:   move.w  #Lenght/4-2,d3 * '...' und CR
179:   jsr     Out(pc)        * ausgeben.
180: * Speicher freigeben und Dos-Lib. schließen
181:   lea     Zwischen+4(pc),a5
182:   moveq   #1,d7
183:   neg.w   d7
184: abbruch  cmp.w   #2,d7
185:   beq     ende
186:   move.l  -(a5),a1
187:   move.l  laenge(pc),d0
188:   move.l  ExecBase,a6
189:   jsr     FreeMem(a6)
190:   addq.w  #1,d7
191:   bra     abbruch
192:   move.l  dosbase(pc),a1
193:   move.l  ExecBase,a6
194:   jsr     CloseLib(a6)
195: ende     moveq   #0,d0
196:   rts   * endlich fertig !!!
197: * Berechnung eines Summanden der Taylorreih
    e von arctan
198: * (Summand=a5, Zwischensp.=a4, länge=d5, Ne
    nner=d6, i=d7)
199: div  move.l  d5,d4 * Länge in Worten-1
200:   subq.l  #1,d4  * nach d4
201:   swap    d4
202:   move.l  a5,a1  * Zeiger auf Summanden
203:   move.l  a4,a0  * Zeiger auf Zwischensp.
204:   moveq   #0,d0
205:   moveq   #0,d1
206:   cmp.l   #65535,d7
207:   bhi     div2
208: loop1    swap    d4
209: loop2    move.w  (a1),d0      * Wort holen
210:   divu    d6,d0         * durch d6 teilen
211:   move.w  d0,(a1)+      * und zurückschreiben
212:   move.w  d0,d1         * denn gleichen Wert
213:   divu    d7,d1         * durch d7 (=i) teilen
214:   move.w  d1,(a0)+      * und in Zwischensp.
215:   dbra    d4,loop2      * doppelte Schleife, da
216:   swap    d4            * dbra nur mit 16-Bit
217:   dbra    d4,loop1      * Zählern arbeitet
218:   rts
219: div2 movem.l d3/d5,-(sp)  * Wenn i>65535,
220:   moveq   #15,d2    * muß umständlicher und
221:   swap    d7        * zeitaufwendiger Divid.
222: Check    btst.l  d2,d7 * werden.
223:   dbne    d2,Check      * Nenner für erste
224:   swap    d7            * Schätzung auf 16 Bit
225:   addq.w  #1,d2         * vermindern.
226:   move.l  d7,d3
227:   lsr.l   d2,d3
228:   addq.w  #1,d3   * Verminderten Nenner
229:   bcc     Ok        * etwas größer wählen,
230:   move.w  #$8000,d3 * damit der gesuchte
231:   addq.w  #1,d2    * Faktor eher zu
232: Ok swap    d2      * klein wird
233:   move.w  d3,d2   * und beides in a2
234:   move.l  d2,a2   * sichern
235: loop21 swap    d4      * Eigentliche
236: loop22 move.w  (a1),d0 * Divisionsschleife
237:   divu    d6,d0      * erste Division wie oben
238:   move.w  d0,(a1)+
239:   move.l  d1,d2      * Faktor abschätzen,
240:   move.l  a2,d3      * durch dividieren
241:   divu    d3,d2      * mit vermindertem
242:   move.w  d2,d5      * 16 Bit Nenner
243:   swap    d5
244:   move.w  d0,d2
245:   divu    d3,d2
246:   move.w  d2,d5
247:   swap    d3      * und zusätzlich
248:   lsr.l   d3,d5   * verschieben
249:   move.l  d7,d3
250:   swap    d3
251:   mulu    d5,d3   * echtes i mit
252:   move.w  d7,d2   * abgeschätztem Faktor
253:   mulu    d5,d2   * multiplizieren
254:   sub.w   d2,d0
255:   move.w  #0,d2
256:   swap    d2
257:   addx.l  d2,d3   * und vom Ausgangswert
258:   sub.l   d3,d1   * abziehen.
259:   swap    d1      * Rest volständig nach
260:   move.w  d0,d1   * d1 schreiben.
261: Korr cmp.l d7,d1  * eventuell das ganze
262:   blt     NoKorr  * noch korregieren
263:   addq.l  #1,d5   * (Rest befindet sich
264:   sub.l   d7,d1   * in d1 (long !) und
265:   bra     Korr    * Faktor in d5 (wort !))
266: NoKorr move.w  d5,(a0)+ * Faktor -> Zwischen.
267:   dbra    d4,loop22
268:   swap    d4
269:   dbra    d4,loop21
270:   movem.l (sp)+,d3/d5 * Register herstellen
271:   rts              * Division beendet
272: * Summand zum Ergebnis addieren
273: * (a3 Zeiger auf Flag, (Add. bzw. Subtr.))
274: summiere move.l  laenge(pc),d4
275:   move.l  Zwischen(pc),a0 * Zeiger auf Ende
276:   adda.l  d4,a0   * Zwischenspeicher
277:   move.l  Summe(pc),a1 * Zeiger auf Ende
278:   adda.l  d4,a1      * Summe
279:   lsr.l   #2,d4      * Laenge in
280:   subq.l  #1,d4      * Langworten - 1
281:   swap    d4
282:   move.w  (a3),d0
283:   eor.w   d7,d0
284:   btst    #1,d0      * jeder 2. Summanden
285:   bne     subtr      * muß subtr. werden
286:   sub     d0,d0      * x-Flag loeschen
287: addi1 swap    d4
288: addi2 addx.l  -(a0),-(a1) * Summe+Summand
289:   dbra    d4,addi2
290:   swap    d4
291:   dbra    d4,addi1
292:   rts
293: subtr    sub  d0,d0 * x-Flag loeschen
294: subtr1   swap d4
295: subtr2 subx.l -(a0),-(a1) * Summe-Summand
296:         dbra    d4,subtr2
297:   swap    d4
298:   dbra    d4,subtr1
299:   rts
300: * Ausgabe eines Wortes in d0 (d0<10000) als
     Dezimalzahl
301: print  moveq   #3,d1  * 4 Ziffern
302:   moveq   #'0',d2     * Ascii-Code von '0'
303:   moveq   #10,d4      * Divisor
304:   adda.l  #4,a5
305: Ziffer divu d4,d0 * Ziffer abspalten
306:   swap    d0
307:   add.w   d2,d0 * in Ascii-Code wandeln
308:   move.b  d0,-(a5) * in Puffer schreiben
309:   move.w  #0,d0
310:   swap    d0
311:   dbra    d1,Ziffer * das Ganze 4 mal
312:   adda.l  #4,a5     * Zeiger erhöhen
313:   dbra    d3,NotOut * Wenn voll, dann...
314: Out move.l  handle(pc),d1  * Puffer ausgeben
315:   lea     Ausgabe(pc),a5 * Pufferanfang
316:   move.l  a5,d2
317:   ext.l   d3  * Pufferlänge berechnen
318:   lsl.l   #2,d3
319:   sub.l   #Lenght-4,d3
320:   neg.l   d3
321:   move.l  dosbase(pc),a6
322:   jsr     Write(a6)
323:   move.l  #Lenght/4-1,d3
324: NotOut   rts  * ... sonst nicht

325: * ----- Datenteil -----

326: dosbase  dc.l 0
327: handle   dc.l 0

328: dosname  dc.b 'dos.library',0
329: * Startwerte: Multiplikator, 1/Argument, Flag
330: Startwert dc.w 48,18,1  *  48*arctan (1/18)
331:   dc.w 32,57,1  *  32*arctan (1/57)
332:   dc.w 20,239,-1 * -20*arctan (1/239)
333: DezLen   dc.l 0     * Anzahl der Stellen
334: laenge   dc.l 0     * Länge in Bytes
335:                     * Komma, nach 1. Wort
336: Summand  dc.l 0     * Zeiger auf jeweiligen
337: Summe    dc.l 0     * Speicherbereich
338: Zwischen dc.l 0
339: Ausgabe  ds.b Lenght * Ausgabepuffer
```


»Pi.asm«:Ein schnelles Programm, um Pi möglichst genau zu berechnen

Programmierkurs Oberon

Auf Wirth'schen Spuren

Oberon, eine Programmiersprache aus dem Wirth'schen Hause, faßt auf dem Amiga langsam aber sicher Fuß. Wo liegen die Stärken der Sprache? Wir bringen Ihnen Oberon in einem umfassenden Ein-/Aufsteigerkurs näher.

von Kai Bolay

Amiga-Oberon ist ein interessanter Compiler dank seines mächtigen Sprachkonzepts. Die Ideen von Modula-2 wurden weiterentwickelt und erweitert [1]. Einige weniger gebräuchliche, hinderliche oder auch für den Compiler-Bauer aufwendige Features entfielen.

Um die in diesem Kurs besprochenen Programme und theoretischen Grundlagen in die Praxis umzusetzen, sollten Sie den Amiga-Oberon-Compiler der A+L AG besitzen [2]. Aber auch mit der Demo-Version [3] lassen sich die ersten Schritte nachvollziehen. Sehr empfehlenswert ist der Source-Level-Debugger. Mit ihm lassen sich alle Programme Schritt für Schritt nachvollziehen.

Die Demo-Version des Amiga-Oberon-Compilers finden Sie auf unserer Diskette zum Heft (Seite 114) oder auf der AMOK-PD-Diskette 53 [3]. Ab Nummer 36 finden Sie auf der AMOK-Serie weitere Oberon-Programme, die die Leistungsfähigkeit von Amiga-Oberon dokumentieren. Sie eignen sich insbesondere, die Sprache besser auszunutzen und kennenzulernen.

Eleganter Einstieg

Am besten lernt man eine Sprache, indem man sie spricht. So paradox das klingen mag, gilt es auch für Programmiersprachen. Ein beliebtes erstes Programm ist »Hello World«. Sein Aufbau ermöglicht es, das Konzept, die Struktur und Funktionsweise einer Programmiersprache relativ einfach nachzuvollziehen. Listing 1 demonstriert es.

Jedes Programm bzw. Modul beginnt mit dem Schlüsselwort »MODULE«, gefolgt von dem Programmnamen. Dieser darf keine Umlaute bzw. Sonderzeichen enthalten und muß mit einem Buchstaben beginnen. Gleiches gilt bei Oberon auch für Variablen-, Typen- und Konstantennamen. Nach dem obligatorischen Semikolon (u.a. auch von C-Programmen bekannt) folgt oft die Importliste. Im Unterschied zu Modula-2 [1] kann nur unqualifiziert importiert werden – es lassen sich somit nur ganze Module importieren. Im fertigen Programm landen dann aber, dank optimierendem Linken, nur die wirklich benötigten Modul-Funktionen.

Was ist ein Modul? Jedes Programm in Oberon ist ein Modul. Unser Hello-World-Programm ist auch eins. Erst durch das Linken (Binden) wird es zum Programm. In Modulen faßt man logisch zueinander passende Teile eines Programms zusammen. Wir werden später noch ausführlicher auf dieses Thema eingehen. Da Bildschirmausgaben nicht zum Sprachkern Oberons gehören, muß man sie selbst programmieren. Doch diese Arbeit wurde schon erledigt. Auf der Compiler-Diskette befindet sich das fertige Modul »io«, das Ein- und Ausgaben auf den Bildschirm regelt.

Mit Hilfe des Schlüsselworts »IMPORT« werden die Funktionen anderer schon vorhandener Module zugänglich. Die Importliste besteht aus Modulnamen, getrennt durch Kommata, die ihrerseits wiederum durch ein Semikolon abgeschlossen werden. In unserem Beispiel wird nur ein Modul importiert.

Anschließend beginnt unser kurzes Programm. Dem Compiler wird dies mit dem Wort »BEGIN« angezeigt. Nun rufen wir die Prozedur »io.WriteString« eines anderen Moduls auf. Eine Prozedur ist ein Unterprogramm und kennzeichnet sog. imperative Programmiersprachen. Eine Prozedur läßt sich von mehreren Programmteilen aufrufen und erlaubt die Zerlegung großer Programme in kleine überschaubare Teile.

Woran erkennt nun der Compiler, aus welchem Modul die Prozedur stammt? Wir müssen es ihm kenntlich machen. Im allgemeinen geschieht dies durch Voranstellen des Modulnamens, also »Modulname.Prozedur«. In unserem Beispiel demzufolge »io.WriteString«. Ruft man eigene Prozeduren innerhalb eines Moduls/Programms auf, entfällt selbstverständlich die Angabe des Modulnamens. Auf die Programmierung eigener Prozeduren gehen wir noch ein.

Die in dem Beispielprogramm verwendete Prozedur WriteString dient einzig dem Zweck, eine Zeichenkette auf dem Bildschirm auszugeben. Diese müssen wir als Parameter übergeben. Ein Parameter ist sozusagen die Eingabe einer Prozedur. Hierüber teilt man ihr mit, welche Daten zu verwenden sind. In Oberon werden die Parameter nach dem Prozedurnamen in Klammern (»(« und »)«) angegeben. Mehrere Parameter trennen wir durch Kommata. io.WriteString benötigt nur einen. Es ist die Zeichenkette, die auszugeben ist; Oberon erwartet sie in Anführungszeichen. Im Beispiel geben wir zusätzlich die Zeichenfolge »\n« an, was nichts anderes als einen Zeilenvorschub bedeutet. Nach der schließenden Klammer folgt wieder ein Semikolon. Neben dem »\n« gibt's noch mehr Steuerzeichen. Diese finden Sie in unserem Kasten.

Hiermit ist das Programm beendet. Oder doch nicht? Wir wissen schon, daß eine Pro-

Amiga-Oberon

Die Umsetzung von Oberon auf den Amiga wurde von Fridtjof Siebert vorgenommen. Oberon umfaßt bereits einen Garbage-Collector (Abfall-Sammler), um Programmierer von der mitunter diffizilen Aufgabe, dynamisch allozierten Speicher freizugeben, wirksam zu entlasten. Garbage-Collection ist eigentlich eine Aufgabe des Betriebssystems.

Nach Jahren praktischer Erfahrung mit Oberon und einer unveröffentlichten objektorientierten Erweiterung entstand 1991 Oberon-2 als universell verwendbare Sprache, die das Konzept von Modula-2 und Oberon fortführte. Das Sprachkonzept wurde im wesentlichen um typgebundene Prozeduren erweitert, die in anderen objektorientierten Sprachen als Methoden bezeichnet werden. Oberon-2 ist eine vollwertige objektorientierte Sprache mit den wesentlichen Konzepten der Datenabstraktion durch Klassenbildung, der Polymorphie und Vererbung durch Typerweiterung und der dynamischen Bindung von Methoden zur Laufzeit. Trotz der Bereicherungen liegt Oberon-2 ein frappierend einfaches und leicht verständliches Konzept zu Grunde.

Bei Amiga-Oberon ist der Editor als eine umfassende, frei programmierbare Entwicklungsumgebung ausgelegt. Alles läßt sich von hier aus perfekt steuern. Der Compiler erzeugt rasant sehr schnellen und hochgradig optimierten Code im Standardformat, auf Wunsch auch für höhere Prozessoren. Jeder für die Codegenerierung nur denkbare Sonderfall kann durch Optionen berücksichtigt werden. Der Garbage-Collector (neu in Version 3.0) verrichtet dank Multitasking seine Arbeit im Hintergrund, ohne das System spürbar zu belasten. Natürlich kostet Garbage-Collection CPU-Zeit. Ein Zusatzprodukt ist der interaktive Debugger, der auf Quelltextebene arbeitet und bei Bedarf Programme schrittweise ausführt. Die maschinennahen Sprachelemente und die Möglichkeit, Maschinencode als INLINE-Statement einzufügen, bieten gute Voraussetzungen für die Systemprogrammierung.

Bezugsquelle: A+L AG, Däderiz 61, CH-2540 Grenchen, Tel.. 00 41 (65) 52 03 11, Fax 00 41 (65) 52 03 79

Preis: Oberon 3.0-Compiler: ca. 350 Mark, Source-Level-Debugger: ca. 230 Mark

zedur bzw. Blöcke mit dem Schlüsselwort BEGIN eingeleitet werden. Fehlt da nicht was? Klar, das Gegenstück – es heißt »END«. Hier lernen wir eine wichtige Struktur sogenannter Hochsprachen kennen: das Blockkonzept. Ein Block ist eine in sich abgeschlossene Programmeinheit und wird durch entsprechende Symbole verpackt. In Pascal bzw. Oberon sind es BEGIN und END, in C z.B. die geschweiften Klammern »{« bzw. »}«. Doch keine Regel ohne Ausnahme: Das Ende eines Moduls muß zusätzlich mit einem Punkt versehen sein, also »END.«.

Starten Sie jetzt den Editor »OEd« Ihres Oberon-Compilers oder der Oberon-Demo und tippen Sie Listing 1 ab (ohne die am Zeilenanfang abgedruckten Zeilennummern; sie dienen lediglich der Orientierung). Um das Programm zu starten, sind zwei Schritte nötig: Das Kompilieren und Binden. Wählen Sie aus dem OEd-Menü den Menüpunkt »Execute« aus. Der Compiler wird gestartet und überprüft den eingegebenen Quelltext auf Tipp- oder sonstige Fehler. Erkennt der Compiler einen, meldet er dies, indem der

```
1: MODULE HelloWorld;
2: IMPORT
3:   io;
4: BEGIN
5:   io.WriteString ("Hello World!\n");
6: END HelloWorld.     © 1993 M&T
```

Listing 1: HelloWorld.mod gibt ebendiesen simplen Text aus – »Hello World«

Cursor auf die entsprechende Position im Quelltext gesetzt wird und eine adäquate Fehlermeldung erscheint. Verlief hingegen das Kompilieren erfolgreich, aktiviert OEd selbständig den Linker »OLink« und bindet das Programm mit den notwendigen Modulen, u.a. dem von uns importierten io-Modul. Anschließend führt OEd das Programm aus. Diese Schritte sind auch »zu Fuß« abrufbar. Verwenden Sie hierzu die Menüpunkte »Compile« und »Link«.

Nachdem wir nun kurz in Oberon hineingeschnuppert haben, widmen wir uns der Theorie. Hauptaufgabe eines Programms ist die Datenbearbeitung. Zuvor allerdings sind diese zunächst einmal einzurichten. Daten, die immer den gleichen Wert besitzen und sich auch während der Laufzeit eines Programms ändern, bezeichnet man als Konstanten. Einer Konstanten gibt man einen Namen. Variablen-, Konstanten-, Prozedurnamen etc. müssen einer bestimmten Definition genügen: Am Anfang steht immer ein Buchstabe oder der Unterstrich »_«. Nun dürfen Buchstaben, Zahlen oder weitere Unterstriche folgen. Wichtiger als Konstanten sind zumeist Variablen. Darunter versteht man Platzhalter bzw. Speicherplätze, die ihren Wert ändern können. Es gibt verschiedene Variablen-Typen (s. Kasten).

Die Zahlen-Typen »SHORTINT« bis »LONGREAL« lassen sich leicht ineinander umwandeln. Rechnet man gemischt, also sowohl mit der einen als auch anderen, besitzt das Ergebnis immer den größten vorkommenden Typ. Multiplizieren wir eine Integer-Zahl mit einer LongReal-Zahl, ist das Ergebnis eine LongReal-Zahl. Doch ein Typ läßt sich auch explizit »verkleinern«. Mit »SHORT« degradiert man eine Real- zu einer LongInt-Zahl. In der Praxis werden Sie diese Möglichkeit schätzen lernen.

Das Programm »Kreis.mod« (Listing 2) demonstriert die Verwendung von Variablen und Konstanten. Zugleich lernen wir Kommentare kennen. Kommentare dienen ausschließlich der Dokumentation und Lesbarkeit von Programmroutinen und werden vom Compiler überlesen. Kommentare müssen in Oberon mit den Zeichenfolgen »(*« und »*)« eingeschlossen werden.

Wie gewohnt beginnt das Programm mit dem Modulkopf und den entsprechenden Importen. Doch bei den Importen versteckt sich etwas Neues. Wir importieren zwei Module, »io« und »RealInOut«. »rio:« bedeutet, daß man anstatt »RealInOut.« auch »rio.« schreiben kann, wenn man sich auf dieses Modul bezieht. Diese Abkürzungsmöglichkeit ist sehr praktisch (man denke an Module mit der abenteuerlichen Schreibweise »LongRealConversions«; der Fehlerteufel läßt grüßen).

Anschließend legen wir die Konstante »Pi« fest. Die Konstantendeklaration wird mit dem Schlüsselwort »CONST« eingeleitet. Vor dem Gleichheitszeichen steht der Name der Konstante, danach ihr Wert. Und wieder das obligatorische Semikolon. Weiterhin definieren wir die beiden Konstanten »Vorkomma« und »Nachkomma«. Für das Programm benötigen wir zudem drei Variablen. Wir nennen sie treffenderweise »Radius«, »Umfang« und »Flaeche«. Alle sind vom Typ »REAL«, da in diesen Variablen Dezimalbrüche gespeichert werden. Im übrigen ist es unbedingt empfehlenswert, Variablennamen so zu bezeichnen, daß ihre Funktion schon aus dem Namen ersichtlich ist (das gilt aber nicht unbedingt für kurzlebige Schleifenvariablen).

Nun könnte man auf die Idee kommen, ausschließlich Variablen vom Typ »LONGREAL« zu verwenden. Schließlich ist es möglich, in LongReal-Zahlen den Inhalt aller anderen Variablentypen aufzunehmen. Der Schluß ist durchaus richtig, aber kaum praktikabel. Berechnungen mit Fließkommazahlen sind viel langsamer als vergleichbare Operationen mit z.B. Ganzzahlen. Benötigt man nur ganze Zahlen, sollte unbedingt auf einen »INTEGER«-Typ ausgewichen werden – er ist einfach schneller. Da unser Programm allerdings Fließkommazahlen benötigt, bleibt uns in diesem Fall nichts anderes übrig, als den Real-Typ zu bemühen.

Steuerzeichen	
\n	neue Zeile
\f	Bildschirm löschen/neue Seite
\t	Tabulator
\b	letztes Zeichen löschen
\r	an den Anfang der Zeile springen
\o	Stringende-Zeichen
\\	Der Backslash (\)
\'	einfaches Anführungszeichen (')
\"	doppeltes Anführungszeichen (")
\NNN	Das Zeichen mit der Octalzahl NNN
\xNN	Das Zeichen mit der Hexzahl NN
\[	Der CSI-Code ("\x3F")
\	Zeilenende im Quelltext überlesen

Nach dem schon bekannten io.WriteString folgt das Einlesen des Radius. Dies geschieht mit Hilfe der Prozedur »ReadReal«, die, wie man am vorangestellten »rio.« erkennt, aus dem Modul»RealInOut« importiert wird. Der Parameter dieser Prozedur ist die Variable, in der der eingelesene Wert abzulegen ist. Beachten Sie die Schlüsselwörter »IF« und »THEN END« noch nicht. Wir kommen darauf zurück. Es folgt die Wertzuweisung der restlichen Variablen. Eine Zuweisung kennzeichnet man in Oberon mit der Symbolfolge »:=«. Hier wird der linken Seite der Wert der rechten zugewiesen. Dabei darf es sich um Zahlen, Ausdrücke oder ähnliches handeln.

Für die Ergebnisausgabe werden die Prozeduren WriteString und »WriteReal« ver-

```
 1: MODULE Kreis; (* Umfang und Flächeninhalt *)
 2: IMPORT
 3:   io, rio: RealInOut;
 4: CONST
 5:   Pi = 3.1415926536; (* Die magische Konstante *)
 6:   Vorkomma = 5;
 7:   Nachkomma = 2;
 8: VAR
 9:   (* Die Variablen *)
10:   Radius, Umfang, Flaeche: REAL;
11: BEGIN
12:   (* Vorbereitung *)
13:   io.WriteString
14:         ("Bitte geben sie den Radius des Kreises ein: ");
15:   IF rio.ReadReal (Radius) THEN END; (* Radius einlesen *)
16:   (* Berechnung ... *)
17:   Umfang := 2 * Pi * Radius;
18:   Flaeche := Pi * Radius * Radius;
19:   (* Ausgabe *)
20:   io.WriteString ("Der Umfang beträgt: ");
21:   IF rio.WriteReal (Umfang, Vorkomma, Nachkomma, FALSE)
22:      THEN END;
23:   io.WriteLn;
24:   io.WriteString ("Die Fläche ist:     ");
25:   IF rio.WriteReal (Flaeche, Vorkomma, Nachkomma, FALSE)
26:      THEN END;
27:   io.WriteLn;
28: END Kreis.                          © 1993 M&T
```

Listing 2: Kreis.mod berechnet den Umfang und die Fläche eines Kreises nach Eingabe

wendet. WriteReal benötigt vier Parameter. Der erste ist die REAL-Zahl, die auszugeben ist. Die nächsten beiden Parameter bestimmen die Anzahl der Vor- und Nachkommastellen. Ob wissenschaftliche Darstellung gewünscht ist, gibt der letzte Parameter an. »FALSE« steht für normale Darstellung, »TRUE« für die Darstellung mit Zehnerpotenzen. Und schon haben wir zwei weitere Oberon-Schlüsselwörter kennengelernt: FALSE und TRUE. Sie repräsentieren im Prinzip nur zwei Zahlen: 0 bzw. 1. TRUE verwenden wir immer, wenn eine Bedingung richtig ist.

Auch hier sollten Sie zunächst das »IF« und »THEN END« wiederum nicht beachten. »WriteLn« bewirkt dasselbe wie »io.WriteString ("\n")«, einen einfachen Zeilenvorschub.

Bedingungen, Schleifen und Fallunterscheidungen

Bis jetzt liefen alle Programme von oben nach unten durch. Es gab keine Möglichkeit, Teile des Programms bei bestimmten Bedingungen nicht auszuführen. Dies ändert sich mit der »IF«-Abfrage. Schauen Sie sich hierzu Listing 3, »Meinung.mod«, an. Wiederum importieren wir das Modul io. Außerdem vereinbaren wir eine Ganzzahlvariable mit dem Namen »Eingabe« und geben einen Text auf dem Bildschirm aus. Dann folgt die erste »IF«-Bedingung. Die Prozedur »ReadInt« hat einen Rückgabewert. Sie teilt uns mit, ob eine Zahl eingelesen werden konnte. Ebendies läßt sich mit dem IF-Kommando abfragen. Liefert ReadInt TRUE, hat alles geklappt. Anschließend werden alle Zeilen bis zur ELSE-Anweisung ausgeführt. Schickt ReadInt hingegen FALSE zurück, werden die zwischen ELSE und END stehenden Programmzeilen abgearbeitet. Der ELSE-Teil ist nicht zwingend notwendig. Trifft eine Bedingung nicht zu, wird das Programm einfach nach END fortgeführt.

Das bedeutet in unserem Beispiel, daß die Zeilen 10-18 ausgeführt werden, falls ReadInt klappt, sonst Zeile 20. Die Struktur wird auch durch das Einrücken der entsprechenden Programmteile deutlich. In den Zeilen 10-18 findet sich ein weiteres IF-Konstrukt. Die Funktion läßt sich einfacher veranschaulichen, wenn man »IF« mit »falls«, »ELSIF« mit »ansonsten, falls«, »ELSE« mit »sonst« und »THEN« mit »dann« übersetzt. Die IF-Konstruktion ist immer mit END abzuschließen.

Ähnlich gewichtig wie Bedingungen sind Schleifen. Sie ermöglichen, Programmteile mehrmals nacheinander auszuführen. Oberon kennt drei Schleifenvarianten. Die erste Art bezeichnet man als »offene Schleife«; sie wird mindestens einmal durchlaufen. In Oberon heißt sie »REPEAT UNTIL«.

Variablentypen von Oberon

Typ	Bedeutung	Werte
BOOLEAN	Wahrheitswert	TRUE/FALSE
CHAR	Zeichen	Steuerzeichen+Buchstaben (0X - 0FFX)
BYTE	Speicherstelle	-128 bis 127 oder 0X bis 0FFX
SHORTINT	kleine Ganzzahl	-128 bis 127
INTEGER	Ganzzahl	-32768 bis 32767
LONGINT	große Ganzzahl	-2147483648 bis 2147483647
REAL	reelle Zahl	-9.223317*10^18 bis 9.223317*10^18
LONGREAL	große reelle Zahl	-10^308 bis 10^308
SHORTSET	kleine Menge	maximal 8 Elemente (0 bis 7)
SET	Menge	maximal 16 Elemente (0 bis 15)
LONGSET	große Menge	maximal 32 Elemente (0 bis 31)

Eine andere Schleifenart ist die »abweisende geschlossene Schleife«. Sie muß nicht durchlaufen werden. In Oberon ist dies die »WHILE«-Schleife.

Letztlich bleibt noch die »LOOP«-Schleife, die an jeder beliebigen Stelle verlassen werden kann. Mit ihr lassen sich die zuvor genannten nachbilden. Trotz des schlechten Rufs dieser Schleifenvariante (sie schadet der Übersichtlichkeit und Klarheit des Quelltexts) kommt man hier und da nicht umhin, diese dennoch zu verwenden.

Die Unterschiede der verschieden Schleifenarten lassen sich an einem alltäglichen Beispiel deutlich machen, der Nahrungsaufnahme. Solange man Hunger hat, soll gegessen werden. Das sieht so aus:

```
WHILE Hunger DO
  Essen;
END;
```

Verpackt in eine Repeat-Schleife :

```
REPEAT
  Essen;
UNTIL NOT Hunger;
```

Hier ist zu beachten, daß »NOT« nicht in der Oberon-Sprachbeschreibung enthalten ist, sondern ein Feature von Amiga-Oberon ist. Wer portable Programme schreiben möchte, sollte »~« anstelle von NOT verwenden. Die Funktion ist identisch.

Auf den ersten Blick erscheint die zweite Variante korrekt. Doch was passiert, wenn man schon vor der Schleife keinen Hunger mehr hatte? Bei der WHILE-Schleife wird nicht gegessen, anders bei »REPEAT«: Es wird mindestens einmal gegessen. Das aber führt zu einer Magenverstimmung. In diesem Fall ist also das While-Konstrukt empfehlenswert. Doch auch mit »LOOP« kommt man zum Ziel:

```
LOOP
  IF NOT Hunger THEN
    EXIT;
  END;
  Essen;
END;
```

Ein gutes Beispiel für Schleifen und Bedingungen ist ein Ratespiel mit Zahlen. Der Computer bestimmt eine Zahl, der Anwender muß sie erraten. Nach jedem Versuch erhält man die Information, ob die geratene Zahl größer oder kleiner der Gesuchten ist. Hat man die Zahl erraten, ist das Spiel zu Ende. In Listing 4 finden Sie die entsprechende Implementation in Amiga-Oberon.

Das Programm beginnt wie gewohnt mit dem Modulkopf und den Importen. Nach der Konstanten- und Variablendeklaration folgt das eigentliche Programm. Zuerst wird die zu erratende Zahl ermittelt. Wir verwenden hierfür die Prozedur »RND« aus dem »Random«-Modul. Sie retourniert eine Zahl zwischen 0 und dem übergebenen Parameter minus 1. Um Zahlen zwischen 1 und der Konstanten Max zu erhalten, addieren wir zum Ergebnis von »r.RND(Max)« noch 1. Ansonsten hätten wir lediglich Zahlen von 0 bis Max-1. Das Ergebnis dieser Berechnung wird der Variablen Zahl zugewiesen.

Es folgt der schon bekannte WriteString-Aufruf. Neu ist die Prozedur »WriteInt«, ebenfalls aus dem io-Modul. Der erste Parameter ist die auf dem Bildschirm auszugebende Zahl. Der zweite die Stellenanzahl. Bis hier alles »olle Kamellen«. Der nächste Parameter allerdings ist für uns neu: eine Schleife. Beim Zahlenraten soll so lange ein

```
1: MODULE Meinung;
2: IMPORT
3:   io;
4: VAR
5:   Eingabe: LONGINT;
6: BEGIN
7:   io.WriteString ("Ihre Meinung zu diesem Programm?\n");
8:   io.WriteString ("(1) = Gut\n(2) = Schlecht\n(3) = Naja\n\nMeinung :");
9:   IF io.ReadInt (Eingabe) THEN
10:    IF Eingabe = 1 THEN
11:      io.WriteString ("Wunderbar!\n");
12:    ELSIF Eingabe = 2 THEN
13:      io.WriteString ("Schade!\n");
14:    ELSIF Eingabe = 3 THEN
15:      io.WriteString ("Na und...\n");
16:    ELSE
17:      io.WriteString ("komische Meinung! nur 1-3 eintippen!\n");
18:    END; (* IF Eingabe *)
19:  ELSE
20:    io.WriteString ("Fehler bei ReadInt() !\n");
21:  END; (* IF *)
22: END Meinung.
```


Listing 3: Meinung.mod wartet auf eine Eingabe und reagiert entsprechend mit Hilfe von Fallunterscheidungen

```
 1: MODULE Zahlenraten;
 2: IMPORT
 3:   io, r: Random;
 4: CONST
 5:   Max = 100;
 6:   MaxStellen = 3;
 7: VAR
 8:   Zahl, Tip: LONGINT;
 9: BEGIN
10:   Zahl := r.RND (Max) + 1;
11:   io.WriteString ("Zahlenraten im
                 Bereich von 1-");
12:   io.WriteInt (Max, MaxStellen);
            io.WriteLn; io.WriteLn;
13:   REPEAT
14:     REPEAT
15:       io.WriteString ("Dein Tip: ");
16:     UNTIL (io.ReadInt (Tip)) AND
                (Tip >= 1) AND (Tip <= Max);
17:     IF Tip > Zahl THEN
18:       io.WriteString ("Leider zu
                   groß!\n\n");
19:     ELSIF Tip < Zahl THEN
20:       io.WriteString ("Schade, zu
                   klein!\n\n");
21:     END; (* IF *)
22:   UNTIL Tip = Zahl;
23:   io.WriteString ("Geschafft!\n");
24: END Zahlenraten.          © 1993 M&T
```

Listing 4: Zahlenraten.mod ermittelt mit Hilfe des Zufallgenerators eine Zahl, die zu erraten ist

Tip eingeholt werden, bis die gesuchte Zahl erraten wurde. Dies wird im Programm durch »UNTIL Tip = Zahl« deutlich. Man erkennt, daß UNTIL in Zeile 22 mit REPEAT in Zeile 13 korrespondiert. Das wird auch durch Einrücken der Zeilen 14 bis 21 innerhalb der Schleife deutlich. Diese Zeilen werden so lange wiederholt, bis die Bedingung hinter UNTIL erfüllt (richtig bzw. TRUE) ist. Für unseren Zweck ist REPEAT ideal.

Die zweite Schleife (Zeile 14 bis 16) sorgt dafür, daß ein gültiger Tip eingegeben wird. Es wird so lange eingelesen, bis ReadInt funktioniert hat und der Tip zwischen 1 und Max liegt. Die Und-Verknüpfung wird in Oberon mit dem Schlüsselwort »AND« gekennzeichnet. Es wird deutlich, daß man bei WHILE, UNTIL und IF nicht nur eine Bedingung angeben kann, sondern zudem die Möglichkeit hat, mehrere zu verknüpfen, indem man die Einzelbedingungen durch »OR« (oder), »AND« (und) und »NOT« (nicht) verbindet. AND ist nicht in der Sprachbeschreibung vorgesehen. Wer portabel sein möchte, sollte »&« verwenden.

Werden zwei Bedingungen durch OR verknüpft, ist die Gesamtbedingung bereits mit einer der beiden Einzelbedingungen erfüllt. Bei AND müssen beide Bedingungen erfüllt sein, damit die Einzelbedingung TRUE ergibt. NOT verkehrt eine Bedingung in ihr Gegenteil.

Das Programm ist aber noch nicht zu Ende: Die verbleibenden Zeilen 17 bis 21 sowie Zeile 23 sollte einsichtig sein. Das vorliegende Listing stellt lediglich ein minimales Spiel dar. Ihre Aufgabe ist es, das Spiel zu erweitern. Zählen Sie beispielsweise die Anzahl der Versuche und geben Sie am Schluß eine Bewertung aus. Auch die Obergrenze des Zahlenbereichs ließe sich variabel einrichten und vom Spieler erfragen. Ein Tip: Man erhöht eine Variable um eine Zahl mit der Anweisung

```
Variable := Variable + Zahl;
```

Das läßt sich weiter vereinfachen. Folgende Anweisung erfüllt die gleiche Aufgabe:

```
INC (Variable, Zahl);
```

»INC« läßt sich von »Inkrementieren« ableiten, was soviel wie »erhöhen« bedeutet. Soll eine Zahl nur um den Wert 1 erhöht werden, reicht der Aufruf

```
INC (Variable);
```

Gleiches gilt zum Reduzieren. Die entsprechende Funktion heißt »DEC« und kommt von »dekrementieren«.

Ein weiterer, die Bewertung vereinfachender Befehl, kann mit Hilfe der »CASE«-Anweisung vorgenommen werden. Mit CASE lassen sich viele IF-Befehle ersetzen und so eine einfache Fallunterscheidung vornehmen. Listing 5 (»Note.mod«) verdeutlicht das. Es bewertet die Leistung eines Schülers. Gleichzeitig finden Sie hier ein Beispiel für die LOOP-Schleife. Die Abbruchbedingung steht in Zeile 13, das CASE-Konstrukt in den Zeilen 11 bis 24. Nach dem Schlüsselwort CASE folgt die Variable, auf die sich die Fallunterscheidung bezieht. Nach »OF« schließen sich die verschiedenen Fälle an, eingeleitet mit dem Zeichen »|«. Jeder Fall ließe sich durch »IF Zensur =« bzw. »ELSIF Zensur =« ersetzen. CASE spart also Tipparbeit. Dies wird besonders in Zeile 16 deutlich, in der die Fälle 2, 3 und 4 zu einem zusammengefaßt werden. Ohne CASE müßte es »IF (Zensur >= 2) AND (Zensur <= 4) THEN« heißen. Anzumerken ist, daß in Variablen vom Typ »CHAR« ein Zeichen gespeichert werden kann. »Read« aus dem io-Modul dient zum Einlesen eines Zeichens von der Tastatur.

Prozeduren

Beschäftigen wir uns jetzt mit Prozeduren. Kaum ein Programm kommt ohne sie aus. Sie erleichtern u.a. das Verständnis eines Programms. In einer Prozedur verpackt man Programmteile, die an verschiedenen Stellen des Programms gebraucht werden. Wir haben bisher viele Prozeduren aus dem Modul io verwendet. Stellen Sie sich vor, Sie müßten jedesmal, wenn Sie eine solche Prozedur aufrufen, anstelle des Aufrufs mehrmals die gleichen Schritte wiederholen. Das kostet nicht nur Speicherplatz, auch der Übersichtlichkeit eines Programms schadet es.

```
 1: MODULE Note;
 2: IMPORT
 3:   io;
 4: VAR
 5:   Zensur: LONGINT;
 6: BEGIN
 7:   LOOP
 8:     REPEAT
 9:       io.WriteString ("Bitte Note oder 0
                 (= Ende) eingeben: ");
10:     UNTIL io.ReadInt (Zensur);
11:     CASE Zensur OF
12:     | 0:
13:       EXIT;
14:     | 1:
15:       io.WriteString ("Echt OK!\n");
16:     | 2..4:
17:       io.WriteString ("Normal\n");
18:     | 5:
19:       io.WriteString ("Gefahr,
                     Achtung!\n");
20:     | 6:
21:       io.WriteString ("und
                 tschüß...\n");
22:     ELSE
23:       io.WriteString ("Bitte eine Note
           von 1 bis 6 oder 0 eingeben!\n");
24:     END; (* CASE *)
25:   END; (* LOOP *)
26: END Note.          © 1993 M&T
```

Listing 5: Note.mod demonstriert mit dem CASE-Operator einfache Fallunterscheidungen

Der Sinn einer Prozedur? Stellen Sie sich vor, Sie sollen mehrere Zahlen von der Tastatur einlesen. Dazu kann man io.ReadInt verwenden. Nun möchten Sie jedesmal einen Text ausgeben und, falls beim Einlesen ein Fehler auftritt, das Einlesen wiederholen. Sie könnten in einer REPEAT-Schleife immer die Prozedur io.WriteString und io.ReadInt in Anspruch nehmen. Umständlich? Richtig. Einfacher geht's mit Prozeduren. Vergleichen Sie einmal Listing 6 und 7 miteinander (»OhneProcs.mod« bzw. »MitProcs.mod«). Bei größeren Prozeduren kann man sich die Arbeitsersparnis sicher gut vorstellen.

Bei »OhneProcs.mod« ist alles klar. Es folgen fünf sich extrem ähnelnde aber dennoch unterschiedliche Schleifen. Interessant wird es bei »MitProcs.mod«. Hier wird eine neue

```
 1: MODULE OhneProcs; (* Ungeschickt *)
 2: IMPORT
 3:   io;
 4: VAR
 5:   Jahr, Monat, Tag, Stunde, Minute: LONGINT;
 6: BEGIN
 7:   REPEAT
 8:     io.WriteString ("Bitte Jahr eingeben:   ");
 9:   UNTIL io.ReadInt (Jahr) AND (Jahr >= 0) AND (Jahr <= 2100);
10:   REPEAT
11:     io.WriteString ("Bitte Monat eingeben:  ");
12:   UNTIL io.ReadInt (Monat) AND (Monat >= 1) AND (Monat <= 12);
13:   REPEAT
14:     io.WriteString ("Bitte Tag eingeben:    ");
15:   UNTIL io.ReadInt (Tag) AND (Tag >= 1) AND (Tag <= 31);
16:   REPEAT
17:     io.WriteString ("Bitte Stunde eingeben: ");
18:   UNTIL io.ReadInt (Stunde) AND (Stunde >= 0) AND (Stunde <= 23);
19:   REPEAT
20:     io.WriteString ("Bitte Minute eingeben: ");
21:   UNTIL io.ReadInt (Minute) AND (Minute >= 0) AND (Minute <= 59);
22:   (* ... der Rest, der diese Daten verarbeitet *)
23: END OhneProcs.                 © 1993 M&T
```

Listing 6: OhneProcs.mod – umständlicher geht's kaum noch

```
 1: MODULE MitProcs; (* geschickt *)
 2: IMPORT
 3:   io;
 4: VAR
 5:   Jahr, Monat, Tag, Stunde, Minute: LONGINT;
 6: PROCEDURE GetNumber (Str: ARRAY OF CHAR; Min, Max: LONGINT): LONGINT;
 7: VAR
 8:   Number: LONGINT;
 9: BEGIN
10:   REPEAT
11:     io.WriteString (Str);
12:   UNTIL io.ReadInt (Number) AND (Number >= Min) AND (Number <= Max);
13:   RETURN Number;
14: END GetNumber;
15: BEGIN
16:   Jahr   := GetNumber ("Bitte Jahr eingeben:   ", 0, 2100);
17:   Monat  := GetNumber ("Bitte Monat eingeben:  ", 1,   12);
18:   Tag    := GetNumber ("Bitte Tag eingeben:    ", 1,   31);
19:   Stunde := GetNumber ("Bitte Stunde eingeben: ", 0,   23);
20:   Minute := GetNumber ("Bitte Minute eingeben: ", 0,   59);
21:   (* ... der Rest, der diese Daten verarbeitet *)
22: END MitProcs.            © 1993 M&T
```

Listing 7: MitProcs.mod ist die geschickte Umsetzung von »OhneProcs.mod« und ersetzt sich oft wiederholende Funktionen durch Prozeduren

Prozedur »GetNumber« geschaffen. Das erkennt man am Schlüsselwort »PROCEDURE«. Es folgen die Parameter, durch Klammern umschlossen. Es gibt auch Prozeduren ohne Parameter (z.B. io.WriteLn). Die einzelnen Parametergruppen sind durch Semikola getrennt. In unserem Beispielprogramm »MitProcs.mod« existieren für die Prozedur GetNumber die Parameter »Str«, »Min« und »Max«. Dabei hat Str den Typ »ARRAY OF CHAR«, auf den wir später genauer eingehen. Es genügt hier zu wissen, daß darin eine Zeichenkette abgelegt werden kann. Min und Max sind LONGINT-Parameter. Diese können innerhalb der Prozedur wie Variablen behandelt werden. Folgt nach der schließenden Klammer ein Doppelpunkt sowie eine Typbezeichnung, handelt es sich bei der Prozedur um eine Funktion, die ein Ergebnis des angegebenen Typs retourniert. Prozeduren liefern von sich aus keine Rückgabewerte.

```
 1: MODULE Raetsel;
 2: IMPORT
 3:   io;
 4: VAR
 5:   a: INTEGER;
 6: PROCEDURE Ping (VAR x: INTEGER);
 7: BEGIN
 8:   DEC (x, 10);
 9: END Ping;
10: PROCEDURE Pong (y: INTEGER);
11: BEGIN
12:   y := y * 2;
13: END Pong;
14: PROCEDURE Zoom;
15: BEGIN
16:   INC (a, 22);
17: END Zoom;
18: PROCEDURE Boom;
19: VAR
20:   a: ARRAY 10 OF CHAR;
21: BEGIN
22:   a := "Hallo!";
23: END Boom;
24: BEGIN
25:   a := 20;
26:   Zoom;
27:   Boom;
28:   Pong (a);
29:   Ping (a);
30:   io.WriteString ("a ist jetzt: ");
31:   io.WriteInt (a, 2);
32:   io.WriteLn;
33: END Raetsel.      © 1993 M&T
```

Listing 8: Raetsel.mod – welchen Wert besitzt die globale Variable a am Programmende?

Es schließen sich, je nach Bedarf, die Konstanten-, Typen- und Variablendeklarationen an, die nur für diese Prozedur gelten und sichtbar sind. Die Parameter innerhalb einer Prozedur lassen sich ohne Auswirkungen (Seiteneffekte) auf übrige Programmteile verändern.

Es gibt aber noch einen weiteren Parametertyp. Ein Beispiel ist io.ReadInt. Diese Funktion verändert den Inhalt der übergebenen Variable. Das läßt sich auch in eigenen Prozeduren bewerkstelligen, indem dem Parameternamen ein »VAR« vorangestellt wird. Wird ein VAR-Parameter innerhalb einer Prozedur/Funktion modifiziert, ändert sich auch der Inhalt der Variablen, die übergeben wurde.

In Prozeduren ist es möglich, auf Variablen des umgebenden Moduls zuzugreifen und diese zu verändern. Ein Sonderfall ist allerdings zu beachten: Besitzt eine lokale Variable den gleichen Namen wie eine globale, dann ist die globale Variable innerhalb der Prozedur unsichtbar und unerreichbar. Sie läßt sich weder auslesen noch verändern, denn der Compiler bezieht sich immer auf die lokale Variable.

Der Prozedurkörper zwischen den Schlüsselwörtern BEGIN und END beinhaltet als Neuerung eigentlich nur noch »RETURN«. RETURN verläßt eine Prozedur. Ist die Prozedur eine Funktion, ist RETURN, gefolgt vom Ergebnis, notwendig. Ansonsten steht es alleine. Das Ergebnis einer Funktion (Prozedur mit Rückgabewert) läßt sich einer Variablen zuweisen.

Man sieht, daß Prozeduren kleine, in sich abgeschlossene und autonome Programmteile sind, die ihre eigenen Variablen und Typen haben können. Außerdem ist es möglich, auf alle Elemente des Moduls zuzugreifen. Prozeduren dürfen auch innerhalb von Prozeduren auftreten, also geschachtelt werden: Sie sind dann allerdings nur innerhalb der umschließenden Prozedur sichtbar. Mit ihrer Umwelt treten Prozeduren durch Parameter und Rückgabewert in Verbindung.

Bringen wir nun unser Beispiel »MitProcs.mod« zu Ende: Im BEGIN-Teil des Moduls wird die Prozedur fünfmal mit den passenden Parametern aufgerufen. Es ist zu beachten, daß die Ausführung des Programms weiterhin mit dem BEGIN-Teil des Moduls startet. Prozeduren werden nur ausgeführt, wenn man sie aufruft, auch wenn sie vor dem BEGIN-Block stehen.

Fassen wir zusammen: Es existieren zwei Parametertypen: Einfache Parameter und VAR-Parameter. Ein Beispiel für einfache Parameter ist die Prozedur Hi:

```
PROCEDURE Hi (a: INTEGER);
BEGIN
  TuDies;
  TuDas;
END Hi;
```

Ruft man sie mit »Hi (42);« auf, passiert dies:

1. a := 42;
2. TuDies;
3. TuDas;

Eine Beispielprozedur für VAR-Parameter ist:

```
PROCEDURE Bye (VAR c: CHAR);
BEGIN
  MachJenes;
  VersucheDieses;
END Bye;
```

Diese läßt sich z.B. so aufrufen: »Bye (Chr);«, wobei Chr eine Variable vom Typ CHAR ist. Die Ausführung sieht dann so aus:

```
 1: MODULE Multiplikation;
 2: IMPORT
 3:   io;
 4: CONST
 5:   Max = 15;
 6:   Stellen = 3;
 7: VAR
 8:   Tabelle: ARRAY Max, Max OF INTEGER;
 9:   x, y: INTEGER;
10: BEGIN
11:   x := 0;
12:   REPEAT
13:     y := 0;
14:     REPEAT
15:       Tabelle[x, y] := (x+1) * (y+1);
16:       INC (y);
17:     UNTIL y = Max;
18:     INC (x);
19:   UNTIL x = Max;
20:
21:   (* ... *)
22:
23:   x := 0;
24:   REPEAT
25:     y := 0;
26:     REPEAT
27:       io.WriteInt (Tabelle[x, y],
                       Stellen + 1);
28:       INC (y);
29:     UNTIL y = Max;
30:     io.WriteLn;
31:     INC (x);
32:   UNTIL x = Max;
33: END Multiplikation.   © 1993 M&T
```

Listing 9: Multiplikation.mod zeigt, wie in Oberon mit mehrdimensionalen Arrays gerechnet wird

1. c := Chr;
2. MachJenes;
3. VersucheDieses;
4. Chr := c;

Man sieht, daß sich auch die übergebene Variable (»Chr«) ändert, wenn man innerhalb der Prozedur »c« modifiziert.

Ein kleines Rätsel: Was gibt das Programm »Raetsel.mod« (Listing 8) aus? Es ist zu beachten, daß nur die Änderung von VAR-Parametern Auswirkungen auf das umge-

bende Programm hat. Tauchen in Prozeduren Variablen mit dem gleichen Namen wie im Modul auf, gelten für die Prozedur die eigenen Variablen.

Nachdem die Importe, Variablen und Prozeduren vereinbart wurden, wird als erste Anweisung im BEGIN-Teil a der Wert 20 zugewiesen. Anschließend rufen wir »Zoom« auf. Diese erhöht den Wert der globalen Variable a um 22. a hat jetzt den Wert 42. Man sieht, daß sich in Prozeduren globale Variablen problemlos ändern lassen.

Jetzt folgt »Boom«. Sie verfügt über eine lokale Variable, deren Name mit der einer globalen identisch ist. Lokal bedeutet, daß die Variable nur für diese Prozedur bestimmt ist. Eigentlich sollte man eine solche Programmierung vermeiden, da sie zu unerwünschten Verwechslungen führen kann. Die neue lokale Variable ist von der globalen völlig unabhängig. Demzufolge ist es auch möglich, ihr einen anderen Typ als der globalen zu geben. Die globale a behält also seinen Wert 42, auch wenn die lokale Variable a auf »Hallo!« gesetzt wird.

Wir sind noch nicht zu Ende. Es folgt der Aufruf der Prozedur »Pong«. Ihr übergeben wir den Wert der globalen Variablen a, also 42. Innerhalb der Prozedur wird der übergebene Parameter verdoppelt. Dies hat aber keinerlei Auswirkungen aufs Hauptprogramm. Der Parameter y existiert nur innerhalb der Prozedur. a behält den Wert 42.

Anders ist dies bei »Ping«. Sie besitzt den VAR-Parameter x. Wir wissen, daß VAR dafür sorgt, daß sich Änderungen an diesem Parameter auch auf das Hauptprogramm auswirken. In unserem Fall bedeutet es, daß, wenn x geändert wird, auch a den neuen Wert übernimmt. Hier werden also x und a um den Wert 10 vermindert und besitzen beide den Wert 32. Damit ist das Rätsel gelöst und das Programm gibt 32 als Wert für a aus.

Weitere Basistypen

Bis jetzt wurden nur die einfachen numerischen Datentypen besprochen. Es existieren aber noch weitere. Man kann diese sogar kombinieren. Um ein Zeichen zu speichern, bietet sich der Typ CHAR an. Dann gibt es

```
 1: MODULE Open;
 2: IMPORT
 3:   io;
 4: PROCEDURE esreveR (Str: ARRAY OF CHAR);
 5: VAR
 6:   i: LONGINT;
 7: BEGIN
 8:   i := LEN (Str)-1; (* Nullbyte
           (Stringende) nicht ausgeben *)
 9:   WHILE i >= 0 DO
10:     io.Write (Str[i]); (* Ein Zeichen
                          ausgeben *)
11:     DEC (i);
12:   END;
13:   io.WriteLn;
14: END esreveR;
15: BEGIN
16:   esreveR ("Hallo Leute!");
17:   esreveR ("Oberon ist einfach toll!");
18:   esreveR ("Was ist wohl 'esreveR'?");
19: END Open.      © 1993 M&T
```

Listing 11: Open.mod arbeitet mit offenen Arrays

```
 1: MODULE ZahlenratenDeLuxe;
 2: IMPORT
 3:   io, r: Random;
 4: VAR
 5:   Zahl, Tip, Max, Versuche: LONGINT;
 6:   jn, dummy: CHAR;
 7: BEGIN
 8:   io.WriteString ("Zahlenraten!\n\n");
 9:   REPEAT
10:     REPEAT
11:       io.WriteString ("Bitte oberen Grenze eingeben: ");
12:     UNTIL io.ReadInt (Max) AND (Max > 1);
13:     Zahl := r.RND (SHORT (Max)) + 1;
14:     Versuche := 0;
15:     REPEAT
16:       REPEAT
17:         io.WriteString ("Dein Tip: ");
18:       UNTIL (io.ReadInt (Tip)) AND (Tip >= 1) AND (Tip <= Max);
19:       INC (Versuche);
20:       IF Tip > Zahl THEN
21:         io.WriteString ("Leider zu groß!\n\n");
22:       ELSIF Tip < Zahl THEN
23:         io.WriteString ("Schade, zu klein!\n\n");
24:       END; (* IF *)
25:     UNTIL Tip = Zahl;
26:     io.WriteString ("Geschafft in"); io.WriteInt (Versuche, 3);
27:     io.WriteString (" Versuchen!\n");
28:     CASE Versuche OF  (* ungerecht, da Max sich ändern kann. Na und!! *)
29:     | 1..3:
30:       io.WriteString ("Voll goil, ey!\n");
31:     | 4..10:
32:       io.WriteString ("Gut, Mann/Frau!\n");
33:     | 11..20:
34:       io.WriteString ("Naja...\n");
35:     ELSE
36:       io.WriteString ("Hahahahaha, üben!!!\n");
37:     END; (* CASE *)
38:     REPEAT
39:       io.WriteString ("\nNochmal (j/n)? ");
40:       io.Read (jn);
41:       jn := CAP (jn); (* in Großbuchstaben umwandeln *)
42:       io.Read (dummy); (* nochmal, da <RETURN> überlesen werden muß *)
43:     UNTIL (jn = "J") OR (jn = "N");
44:   UNTIL jn = "N";
45: END ZahlenratenDeLuxe.        © 1993 M&T
```

Listing 10: ZahlenratenDeLuxe.mod ist eine verbesserte Version von Listing 4, »Zahlenraten.mod«

noch den Außenseiter »BYTE«, der vor allem beim Zugriff auf das Betriebssystem von Bedeutung ist. Eine Sonderstellung nimmt der Typ »ARRAY OF BYTE« ein, der zu allen anderen Array-Typen kompatibel ist. Auf Arrays gehen wir noch genauer ein.

Variablen, die als »BOOLEAN« definiert wurden, können Wahrheitswerte speichern. Diese kennen nur zwei Zustände: TRUE für wahr und FALSE für unwahr bzw. falsch. Der Zweck solcher Variablen wird an einem Beispiel deutlich. Ein Programm fragt am Anfang den Benutzer, ob er englische oder deutsche Benutzerführung will. Die BOOLEAN-Variable »Deutsch« wird mit »:=« entsprechend belegt. Mit IF läßt sich eine Fallunterscheidung vornehmen:

```
IF Deutsch
  THEN
    io.WriteString ("Apfel");
  ELSE
    io.WriteString ("Apple");
END;
```

Als weiteren Basistyp gibt es noch die Gruppe der Sets (»SHORTSET«, »SET« und »LONGSET«). Sie repräsentieren jeweils eine Menge mit 8, 16 oder 32 Elementen. Das läßt erahnen, daß sich eine Menge also aus mehreren Elementen zusammensetzt. Wer in der Grundschule Mengenlehre hatte, wird sich hier heimisch fühlen. Jedes einzelne Element ist entweder in der Menge enthalten oder nicht. Die Elemente werden ab 0 aufwärts durchnumeriert. Es empfiehlt sich, den einzelnen Elementen Namen zu geben, indem man Konstanten vereinbart.

Als Beispiel einer Menge kann eine Schublade dienen, in der sich Gegenstände befinden (oder auch nicht). Die einzelnen Gegenstände werden von 0 an numeriert:

```
CONST
  Geld      = 0;
  Papier    = 1;
  Bleistift = 2;
  Kuli      = 3;
  Buntstift = 4;
  Heft      = 5;
  Besteck   = 6;
```

Anschließend ist die Schublade zu definieren:

```
VAR Schublade: SHORTSET;
```

Um der Schublade einen Inhalt zuzuweisen, schreibt man:

```
Schublade := SHORTSET {
               Heft,
               Buntstift,
               Papier
             };
```

Jetzt befinden sich diese drei Elemente in der Schublade. Man legt weitere Dinge hinein mit:

```
Schublade := Schublade +
          SHORTSET {Geld, Besteck};
```

Um nur ein Element hinzuzufügen:

```
INCL (Schublade, Kuli);
```

Elemente entfernt man mit:

```
Schublade := Schublade -
          SHORTSET {Kuli, Papier};
```

bzw.

```
EXCL (Schublade, Geld);
```

Möchte man wissen, was sowohl in S1 als auch in S2 (beide vom Typ »SHORTSET«) liegt:

```
  1: MODULE Init;
  2: IMPORT
  3:   io;
  4: CONST
  5:   Blau = 0;
  6:   Gruen = 1;
  7:   Violett = 2;
  8: (* ... *)
  9: TYPE
 10:   Feld = RECORD
 11:     name: ARRAY 50 OF CHAR;
 12:     spieler: SET;
 13:   END;
 14:   FeldPtr = POINTER TO Feld;
 15:   Kaeuflich = RECORD (Feld)
 16:     preis: INTEGER;
 17:     hypothek: BOOLEAN;
 18:     gekauft: BOOLEAN;
 19:   END;
 20:   KaeuflichPtr = POINTER TO Kaeuflich;
 21:   Tarife = ARRAY 6 OF INTEGER;
 22:   Strasse = RECORD (Kaeuflich)
 23:     farbe: INTEGER;
 24:     haeuser: INTEGER;
 25:     miete: Tarife;
 26:   END;
 27:   StrassePtr = POINTER TO Strasse;
 28:   Bahnhof = RECORD (Kaeuflich)
 29:   END;
 30:   BahnhofPtr = POINTER TO Bahnhof;
 31:   (* ... weitere Typen ... *)
 32: VAR
 33:   Plan: ARRAY 40 OF FeldPtr;
 34: PROCEDURE Kaufen (VAR KaufFeld: FeldPtr);
 35: BEGIN
 36:   IF KaufFeld IS Kaeuflich THEN (*
                             Typtest !!! *)
 37:     WITH KaufFeld: Kaeuflich DO
 38:       IF KaufFeld.gekauft THEN
 39:         io.WriteString (KaufFeld.name);
 40:         io.WriteString (" ist schon
                            verkauft!\n");
 41:       ELSE
 42:         KaufFeld.gekauft := TRUE;
 43:         (* ... *)
 44:       END;
 45:     END;
 46:   ELSE
 47:     io.WriteString (KaufFeld.name);
 48:     io.WriteString (" kann nicht gekauft
                             werden!\n");
 49:   END;
 50: END Kaufen;
 51: PROCEDURE InitFeld
          (Name: ARRAY OF CHAR): FeldPtr;
 52: VAR
 53:   NeuesFeld: FeldPtr;
 54: BEGIN
 55:   NEW (NeuesFeld); (* Speicher holen *)
 56:   IF NeuesFeld = NIL THEN HALT (20) END; (*
kein Speicher! *)
 57:
 58:   COPY (Name, NeuesFeld.name);
 59:   NeuesFeld.spieler := SET {};
 60:   RETURN NeuesFeld;
 61: END InitFeld;
 62: PROCEDURE InitStrasse
          (Name: ARRAY OF CHAR;
 63:       Preis: INTEGER;
 64:       Farbe: INTEGER;
 65:       Miete: Tarife): StrassePtr;
 66: VAR
 67:   NeueStrasse: StrassePtr;
 68: BEGIN
 69:   NEW (NeueStrasse); (* Speicher holen *)
 70:   IF NeueStrasse = NIL THEN HALT (20)
                 END; (* kein Speicher! *)
 71:   COPY (Name, NeueStrasse.name);
 72:   NeueStrasse.spieler  := SET {};
 73:   NeueStrasse.preis    := Preis;
 74:   NeueStrasse.hypothek := FALSE;
 75:   NeueStrasse.gekauft  := FALSE;
 76:   NeueStrasse.farbe    := Farbe;
 77:   NeueStrasse.haeuser  := 0;
 78:   NeueStrasse.miete    := Miete;
 79:   RETURN NeueStrasse;
 80: END InitStrasse;
 81: PROCEDURE InitBahnhof
          (Name: ARRAY OF CHAR;
 82:       Preis: INTEGER): BahnhofPtr;
 83: VAR
 84:   NeuerBahnhof: BahnhofPtr;
 85: BEGIN
 86:   NEW (NeuerBahnhof);(* Speicher holen *)
 87:   IF NeuerBahnhof = NIL THEN HALT (20)
                 END; (* kein Speicher! *)
 88:   COPY (Name, NeuerBahnhof.name);
 89:   NeuerBahnhof.spieler  := SET {};
 90:   NeuerBahnhof.preis    := Preis;
 91:   NeuerBahnhof.hypothek := FALSE;
 92:   NeuerBahnhof.gekauft  := FALSE;
 93:   RETURN NeuerBahnhof;
 94: END InitBahnhof;
 95: (* ... *)
 96: BEGIN
 97:   Plan[0] := InitFeld ("Start");
 98:   Plan[1] :=
       InitStrasse ("Oberonweg", 333, Blau,
 99:                           Tarife (10, 50,
100, 150, 200, 500));
100: (* ... *)
101:   Plan[5] := InitBahnhof
                  ("Hauptbahnhof", 500);
102: (* ... *)
103:   Kaufen (Plan[0]);
       Kaufen (Plan[1]); Kaufen (Plan[1]);
104: END Init.    © 1993 M&T
```

Listing 12: Init.mod – Pointer, Records und sonstige üble Dinge

```
InBeiden := S1 * S2;
```

Als letzten Mengenoperator gibt es noch:

```
InEiner := S1 / S2;
```

Mit »/« erhält man alle Elemente, die in S1 oder S2 sind, aber nicht in beiden gleichzeitig. Dabei ist unbedingt zu beachten, daß sich in Mengen nur feststellen läßt, ob ein Element enthalten ist. Ein spezielles Element kann also nicht mehrmals vorkommen. Zweimalige Anweisung von »INCL (S1, 2)« bewirkt dasselbe wie nur eine. Anschließend ist das Element »2« in der »S1« enthalten.

Häufig tritt das Problem auf, viele gleichartige Daten zu speichern (z.B. Tabellen). Für eine zehnspaltige Tabelle ließen sich mit

```
VAR a, b, c, d, e, f, g, h, i, j: INTEGER;
```

die passenden Variablen bereitstellen. Umständlich. Das wird spätestens dann deutlich, wenn man die Tabelleneinträge aufaddieren möchte. Bei zehn Werten mag es noch möglich sein, doch was ist bei 1 000 oder mehr?

Die Lösung sind sog. Arrays. Sie dienen dazu, viele gleichartige Variablen in einer Tabelle zusammenzufassen. Die Syntax der Variablendefinition ist:

```
VAR Tab: ARRAY 1000 OF INTEGER;
```

Damit hat man auf einen Schlag 1000 Variablen definiert. Die erste davon heißt »Tab[0]«, die letzte »Tab[999]«. Um alle Elemente dieser Tabelle zu addieren, schreibt man:

```
Summe := 0;
Nummer := 0;
WHILE Nummer < 1000  DO
  INC (Summe, Tab[Nummer]);
  INC (Nummer);
END;
```

Man sieht, Arrays und Schleifen vertragen sich prima.

Da aber auch häufig Tabellen mit mehreren Spalten benötigt werden, muß es weitere Definitionsmöglichkeiten geben. Oberon beherrscht auch das. Listing 9 (»Multiplikation.mod«) demonstriert die Arbeit mit mehrdimensionalen Arrays. Zunächst legen wir eine Tabelle mit 15 x 15 Elementen an. Die Konstante »Max« wurde zuvor mit dem Wert 15 besetzt. Man sieht: Die Dimension wird nach dem Schlüsselwort »Array« angegeben. Gibt es mehrere, sind diese durch Kommata zu trennen. Anschließend füllen wir die Tabelle mit Werten und geben sie auf dem Bildschirm aus.

Kommen mehrere gleichartige Arrays in einem Modul vor oder möchte man einem Array einen vielsagenden Namen geben, legt man einfach einen neuen Typ an:

```
TYPE Tab = ARRAY 10 OF INTEGER;
```

Dieser läßt sich nun bei der Variablendeklaration einsetzen:

```
VAR Alter, Groesse: Tab;
```

Man könnte auch schreiben

```
VAR Alter, Groesse: ARRAY 10 OF INTEGER;
```

Der Vorteil der ersten Schreibweise mit eigenem Typ ist die, daß sich alle Variablen dieses Typs problemlos einander zuweisen lassen.

Im übrigen sind Zeichenketten (Strings) auch Arrays, und zwar vom Typ CHAR. Die Definition geschieht also mit »ARRAY OF CHAR«. Einer so deklarierten Variablen kann mit Hilfe des Zuweisungssymbols »:=« ein in Anführungszeichen eingeschlossener String übergeben werden. Auf die Buchstaben ist, wie bei Arrays üblich, mit »[]« zuzugreifen. Möchte man also das 13. Zeichen der Zeichenkette »Str« in Erfahrung bringen, genügt die Anweisung Str[12]. Im letzten Element der Zeichenkette befindet sich ein Null-Byte (»\0«) als Ende-Kennung. Gerade bei der Programmierung des Amiga-Betriebssystems ist es besonders wichtig, Zeichenketten mit einem Null-Byte abzuschließen.

Anders als z.B. bei BASIC sind Befehle zur Manipulation von Zeichenketten nicht im Sprachkern von Oberon enthalten. Das muß nicht unbedingt ein Nachteil sein, denn die benötigten Prozeduren stehen als Modul bereit: es heißt »Strings« und liegt dem Compiler bei.

Eine weitere Besonderheit sind offene Arrays. Bis jetzt kennen wir nur solche mit einer definierten Länge: sie haben wir bei der Variablendefinition angegeben. Offene Arrays verfügen über keine bestimmte Länge und existieren nur in Verbindung mit Prozedur-Parametern. Im Prozedurkopf läßt sich als Parametertyp ein Array ohne Größenangabe vereinbaren. So lassen sich Arrays beliebiger Größe übergeben. Innerhalb der Prozedur kann die Länge dann mit »LEN (Array)« in Erfahrung gebracht werden. Ein Beispiel für diese Technik sind viele Prozeduren im io-Modul. Ein Beispiel ist Listing 10, »Open.mod«.

Records und Pointer

Neben Arrays kennt Oberon weitere Möglichkeiten, Variablen zu strukturieren und gruppieren: Records. Ein Record faßt mehreren Typen (z.B. logisch zueinander passende Variablen) zu einem neuen zusammen.

Beim Brettspiel »Monopoly« gehört z.B. zu einem Feld ein Name und es muß zudem vermerkt werden, welche Spieler auf diesem Feld stehen. Es bietet sich diese Form an:

```
TYPE
  Feld = RECORD
    name: ARRAY 50 OF CHAR;
```

```
    spieler: SET;
  END;
VAR
  Los: Feld;
```

Schon hat man einen neuen (Record-) Typ geschaffen, der zwei verschiedene Variablen enthält: name und spieler. Der Zugriff auf die einzelnen Strukturelemente erfolgt mit dem Zeichen ».«:

```
io.WriteString (Los.name);
```

Doch die meisten Monopoly-Felder haben weitere Eigenschaften: Felder, die gekauft werden können, haben einen Preis und es kann eine Hypothek aufgenommen werden:

```
 1: MODULE ArgDemo;
 2: IMPORT
 3:   arg: Arguments, io;
 4: VAR
 5:   i: INTEGER;
 6:   str: ARRAY 80 OF CHAR;
 7: BEGIN
 8:   i := arg.NumArgs();
 9:   WHILE i > 0 DO
10:     io.WriteInt (i, 2);
11:     io.WriteString (": ");
12:     arg.GetArg (i, str);
13:     io.WriteString (str);
14:     io.WriteLn;
15:     DEC (i);
16:   END;
17:   io.WriteString ("Das Programm heißt: ");
18:   arg.GetArg (0, str);
19:   io.WriteString (str);
20:   io.WriteLn;
21: END ArgDemo.
```


Listing 13: ArgDemo.mod wertet die CLI/Shell- oder Workbench-Argumente aus

```
TYPE
  Kaeuflich = RECORD (Feld)
    preis: INTEGER;
    hypothek: BOOLEAN;
    gekauft: BOOLEAN;
  END;
```

Schon taucht eine weitere Neuerung auf: Nach dem Schlüsselwort RECORD erscheint ein anderer Record-Typ in Klammern. Das bedeutet, daß die Elemente des Basistyps (so bezeichnet man den Record in Klammern, hier also Feld) ebenfalls im neuen Typ integriert sind. Auf diese Art verpassen wir also käuflichen Feldern einen Namen. Eine Straße und ein Bahnhof lassen sich so einrichten:

```
TYPE
  Tarife = ARRAY 6 OF INTEGER;
  Strasse = RECORD (Kaeuflich)
    farbe: INTEGER;
    haeuser: INTEGER;
    miete: Tarife;
  END;
  Bahnhof = RECORD (Kaeuflich)
  END;
```

»Strasse« beinhaltet weitere Elemente, »Bahnhof« hingegen kennt lediglich die gleichen wie »Kaeuflich«. Daß man trotzdem einen neuen Typ definiert, kommt nicht von ungefähr. Mit Typtests kann man so später in Erfahrung bringen, ob ein Feld ein Bahnhof ist oder nicht.

Der große Vorteil dieser Definition – man bezeichnet das auch als Vererbung von Elementen der Basistypen –, ist, daß sich Prozeduren mit eben solchen Parametern entwickeln lassen. So ließe sie sich mit dem Parameter »Strasse« aufrufen, da Strasse eine Erweiterung von »Kaeuflich« und damit auch von »Feld« ist. Prozeduren mit dem Parameter »Kaeuflich« (z.B. Straße kaufen oder Hypothek aufnehmen) verweigern hingegen die Arbeit bei einfachen Feldern, jedoch nicht bei Straßen oder Bahnhöfen. Wäre es nicht schön, wenn sich mit

```
VAR Plan: ARRAY 40 OF Feld;
```

gleichzeitig alle 40 Variablen für die verschiedenen Feldtypen anlegen ließen? Im Prinzip ja, wäre da nicht ein Stein im Weg, den es wegzuräumen gilt: leider sind nicht alle Felder eines Monopoly-Spiels identisch. »Plan[1]« z.B. ist eine Straße, die sich nicht einfach einer Variable vom Typ »Feld« zuordnen läßt. »Strasse« beinhaltet »Feld«, aber »Feld« nicht »Strasse«.

Einen Ausweg bieten »Pointer« (deutsch: Zeiger). Einige werden sie vielleicht von der Programmiersprache C kennen, in der Pointer zum Alltag gehören. In C allerdings ist die Arbeit mit Pointer nicht immer ungefährlich, da die simplen Compiler viel zu viel fehlerhafte Pointer-Operationen durchgehen lassen. Die Notation von Oberon und der Überprüfungsvorgang des Compilers nehmen den Zeigern jedoch ihre Gefährlichkeit.

Jede Variable wird in einer bestimmten Speicherzelle im Hauptspeicher des Computers abgelegt. Diesen Platz bezeichnet man als die Adresse der Variable. Der Datentyp, in dem Adressen gespeichert werden, sind Zeiger. Einen Zeiger definiert man in Oberon so:

```
VAR StartAdr: POINTER TO Feld;
```

bzw.

```
TYPE FeldPtr = POINTER TO Feld;
VAR StartAdr: FeldPtr;
```

Man erkennt, daß in Oberon Zeiger an einen Typ gebunden sind. In einer Variablen vom Typ »FeldPtr« kann man also nur die Adresse einer Variable vom Typ »Feld« speichern.

Wofür benötigen wir nun aber Zeiger für das Monopoly-Spiel? Wir wissen, daß »Strasse« eine Erweiterung von »Feld« ist. Ein Zeiger aufs »Feld« kann dementsprechend auch auf Variablen vom Typ »Strasse« zeigen. Unser Spielplan sieht jetzt so aus:

```
VAR Plan: ARRAY 40 OF FeldPtr;
```

Doch Vorsicht. Mit dieser Deklaration hat man 40 Variablen definiert, die Adressen von Feldern speichern können. Die Felder selbst hat man damit noch nicht angelegt. Dies muß explizit mit der Funktion »NEW« geschehen, die den benötigten Speicher reserviert, in dem die Elemente abzulegen sind. Es bietet sich an, für jede Feldart eine eigene Prozedur zu programmieren, die die einzelnen Elemente mit Werten belegt. Listing 12 (»Init.mod«) zeigt, wie das funktioniert.

Die Funktion »COPY« in Listing 12 kopiert die Stringvariablen. Eine direkte Zuweisung mit »:=« ist nicht möglich, da die Zeichenkettenvariablen unterschiedliche Längen haben. Bei Copy muß man auf die Reihenfolge der Parameter achten. Sie ist

```
COPY (Quelle, Ziel);
```

was der Anweisung

```
Ziel := Quelle;
```

entspricht.

Ebenfalls neu ist der Gebrauch strukturierter Konstanten. Eine solche verwenden wir beim Aufruf der Prozedur »InitStrasse«. Was auf den ersten Blick wie ein Sprung zur Prozedur »Tarife« aussieht, ist in Wirklichkeit eine strukturierte Konstante vom Typ »Tarife«. Man kann erkennen, daß sich so alle Array- oder Record-Werte in einem Rutsch zuweisen lassen. Vorsicht: Strukturierte Konstanten sind nur mit Amiga-Oberon möglich, der Original-Oberon-Report von N. Wirth sieht sie nicht vor.

In der Prozedur »Kaufen« taucht plötzlich eine »WITH«-Anweisung auf. Mit ihr läßt sich erzwingen, daß eine Variable innerhalb des WITH-Blocks so behandelt wird, als sei sie von dem Typ, der nach dem »:« angegeben ist. Dieser muß allerdings eine Erweiterung vom Typ der Variablen sein, sonst gibt's einen Laufzeitfehler. Per »IS« kann man prüfen, ob eine Variable einen bestimmten Typ hat. Wir tun das in der Zeile vor der WITH-Anweisung.

```
 1: MODULE China;
 2: IMPORT
 3:   io, fs: FileSystem, a: Arguments;
 4: VAR
 5:   In, Out: fs.File;
 6:   Chr: CHAR;
 7:   Arg: ARRAY 80 OF CHAR;
 8: BEGIN
 9:   IF a.NumArgs() # 2 THEN
10:     io.WriteString ("Usage: ");
11:     a.GetArg (0, Arg);
12:     io.WriteString (Arg);
13:     io.WriteString (" InFile/A,
                         OutFile/A\n");
14:     HALT (20);
15:   END;
16:   a.GetArg (1, Arg);
17:   IF NOT fs.Open (In, Arg, FALSE) THEN
18:     io.WriteString ("Can't open ");
19:     io.WriteString (Arg);
20:     io.WriteString (" for input!\n");
21:     HALT (20);
22:   END; (* IF *)
23:   a.GetArg (2, Arg);
24:   IF NOT fs.Open (Out, Arg, TRUE) THEN
25:     io.WriteString ("Can't open ");
26:     io.WriteString (Arg);
27:     io.WriteString (" for output!\n");
28:     HALT (20);
29:   END; (* IF *)
30:   LOOP
31:     IF NOT fs.ReadChar (In, Chr) THEN
          EXIT END;
32:     IF Chr = 'r' THEN Chr := 'l' END;
33:     IF Chr = 'R' THEN Chr := 'L' END;
34:     IF NOT fs.WriteChar (Out, Chr) THEN
          EXIT END;
35:   END;
36:   IF (In.status # fs.eof) OR
       (Out.status # fs.ok) THEN
37:     io.WriteString ("Error processing
                        file!\n");
38:   END;
39:   IF NOT fs.Close (In)  THEN
       io.WriteString ("Close failed!?\n")
40:   END;
41:   IF NOT fs.Close (Out) THEN
       io.WriteString ("Close failed!?\n")
42:   END;
41: END China.
```


Listing 14: China.mod benutzt u.a. Funktionen des Ein- und Ausgabemoduls FileSystem

Eine Variable läßt sich mit Hilfe des »Typeguards« auch ohne WITH erweitern. Schreibt man

```
KaufFeld(Kaeuflich).gekauft
```

wird »Kauffeld« so behandelt, als sei es vom Typ »Kaeuflich«.

Innerhalb der »Init«-Prozeduren wird mit »NEW« der notwendige Speicher besorgt. Hat ein Zeiger nach NEW den Wert »NIL«, ließ sich der Speicher nicht reservieren (möglicherweise zu wenig vorhanden). An dieser Stelle bricht das Programm mit einem Fehler ab (»HALT«). NIL als Zeigerinhalt bedeutet, daß der Zeiger auf keinen Speicherbereich zeigt, also unbenutzt ist.

Im übrigen ist nach jedem Versuch, Speicher zu allokieren, eine entsprechende Überprüfung vorzunehmen. Im schlimmsten Fall (der Speicher ließ sich nicht reservieren) kann es passieren, daß Sie in einen Speicherbereich schreiben, der z.B. von anderen Programmen benötigt wird.

Wie man auf Elemente eines Records über einen Zeiger zugreift, wissen Sie bereits: ein einfacher ».« genügt. Möchte man jedoch alle Elemente einer Variablen erreichen, ist der »^«-Operator zu verwenden. Um den Inhalt bzw. die Daten, auf die der Zeiger zeigt, zu kopieren, schreibt man:

```
(1)  Zeiger2^ := Zeiger1^;
```

Dabei ist zu beachten, daß Zeiger 2 schon auf etwas zeigen muß, also zuvor z.B. mit »NEW« eingerichtet wurde. (1) kopiert nun die Daten, auf die Zeiger 1 zeigt. Ändert man Elemente von Zeiger 2, hat dies keinen Einfluß auf die Daten von Zeiger 1.

Anders verhält es sich bei

```
(2)  Zeiger2 := Zeiger1;
```

Hier wird lediglich die Adresse, nicht der Inhalt der Daten kopiert. Ändert man den Inhalt über Zeiger 2, hat dies direkte Auswirkungen auf den Inhalt von Zeiger 1, denn beide Zeiger beziehen sich auf die selbe Speicheradresse. Den mit »^« eingeleiteten Vorgang bezeichnet man als dereferenzieren, denn ein Zeiger stellt eine Referenz bzw. einen Verweis auf Daten dar.

Benötigt man einen Zeiger nicht mehr, kann, nein muß man, den zuvor allokierten Speicher mit »DISPOSE« freigeben. Dabei ist unbedingt zu beachten, daß alle Zeiger, die auf diese Adresse zeigen, einen undefinierten Inhalt haben. Mit

```
DISPOSE (Zeiger1);
```

läßt sich Zeiger 2 aus (1) weiterhin bedenkenlos verwenden, der in (2) zugewiesene Zeiger 2 ist jedoch ungültig.

Zeiger-Arithmetik ist auch bei Oberon unumgänglich

Doch auch des Guten zuviel ist möglich: Gibt man einen Speicherbereich zweimal mit Dispose frei, läßt der Guru grüßen.

Ein wichtiges Einsatzgebiet für Zeiger sind dynamische Listen. Denken Sie an einen Texteditor: Beim Programmstart ist nicht ersichtlich, wieviele Zeilen benötigt werden. Legt man also z.B. ein statisches Array mit 1024 Zeilen und 256 Spalten an, werden sofort 256 KByte Speicher belegt. Nun kann es aber sein, daß nur 20 Zeilen oder weniger notwendig sind. Oder andersherum: vielleicht sind 1 024 Zeilen sogar zu wenig?

Ein Ausweg aus diesem Dilemma sind Listen. Darunter ist eine Verkettung gleicher Elemente (Strukturen) zu verstehen. Bei einer Liste existiert immer ein Kopf (Head) und ein Ende (Tail). Besteht die Liste nur aus einem Element, ist der Kopf gleich dem Ende. Ansonsten verweist der Kopf auf das nächste Glied, dieses auf ein weiteres usw. Mit Hilfe von Zeigern kann man sich so von Glied zu Glied durchhangeln, bis man irgendwann das Ende der Liste erreicht. Dieses zeigt dann im allgemeinen auf NIL. Um die Verwaltung solcher Listen zu vereinfachen, stellt der Oberon-Compiler das »Lists«-Modul zur Verfügung. Mit Hilfe der darin befindlichen Prozeduren läßt sich u.a. eine Zeile so anlegen (ein Blick in das Modul erleichtert das Verständnis ungemein):

```
IMPORT Lists;
TYPE
  Zeile = RECORD (Lists.Node)
    str: ARRAY 256 OF CHAR;
  END;
  ZeilePtr = POINTER TO Zeile;
VAR
  Liste: Lists.List;
```

Das Schöne ist, daß man weiterhin problemlos die Prozeduren aus »Lists« verwenden kann, da »Zeile« eine Erweiterung von »Node« ist. Je nach Bedarf läßt sich so Speicher für eine Zeile anfordern (»NEW«). Das erzeugte Element wird so in die Liste eingefügt (»AddHead«, »AddTail«, »AddBefore«, »AddBehind«). Möchte man ein Glied der Kette löschen, reicht ein Aufruf der Funktion »Remove« mit anschließendem »DISPOSE«,

```
 1: MODULE MiniPaint;
 2: IMPORT
 3:   I: Intuition, rq: Requests, s: SYSTEM, d: Dos, e: Exec,
      g: Graphics;
 4: CONST
 5:   Version = "$VER: MiniPaint 1.00 (10-Jul-91)\n\r";
 6:   (* Für den Version-Befehl von OS 2.0 ! Example:
        "Version MiniPaint" *)
 7: VAR
 8:   nw: I.NewWindow;
 9:   Win: I.WindowPtr;
10:   IMsg: I.IntuiMessagePtr;
11:   Draw, Quit: BOOLEAN;
12:   Class: LONGSET;
13:   Code: INTEGER;
14:   OldX, OldY, X, Y, Color: INTEGER;
15: PROCEDURE Cut (VAR i: INTEGER; Min, Max: INTEGER);
16: BEGIN
17:   IF i < Min THEN i := Min END;
18:   IF i > Max THEN i := Max END;
19: END Cut;
20: BEGIN
21:   Draw := FALSE; Quit := FALSE; Color := 1;
22:   nw := I.NewWindow (100, 75, 300, 100, 0, 1,
                LONGSET {I.mouseButtons,
23:                      I.mouseMove, I.closeWindow},
                LONGSET {I.rmbTrap,
24:             I.reportMouse, I.windowDrag, I.sizeBRight, I.activate,
25:                      I.windowSizing, I.windowDepth, I.windowClose},
26:                      NIL, NIL, s.ADR ("MiniPaint v1.00"), NIL,
27:                      NIL, 150, 50, -1, -1, SET {I.wbenchScreen});
28:   Win := I.OpenWindow (nw);
29:   rq.Assert (Win # NIL, "Sorry, no Window");
30:   g.SetDrMd (Win^.rPort, g.jam1);
31:   g.SetAPen (Win^.rPort, Color);
32:   REPEAT
33:     e.WaitPort (Win^.userPort);
34:     LOOP
35:       IMsg := e.GetMsg (Win^.userPort);
36:       IF IMsg = NIL THEN EXIT END;
37:       Class := IMsg^.class;
38:       Code := IMsg^.code;
39:       X := IMsg^.mouseX;
40:       Cut (X, Win^.borderLeft+1, Win^.width-Win^.borderRight-2);
41:       Y := IMsg^.mouseY;
42:       Cut (Y, Win^.borderTop+1, Win^.height-Win^.borderBottom-2);
43:       e.ReplyMsg (IMsg);
44:       IF (I.mouseButtons IN Class) THEN
45:         CASE Code OF
46:         | I.selectUp:
47:           Draw := FALSE;
48:         | I.selectDown:
49:           Draw := TRUE;
50:           OldX := X; OldY := Y;
51:           IF g.WritePixel (Win^.rPort, X, Y) THEN END;
52:           g.Move (Win^.rPort, X, Y);
53:         | I.menuUp:
54:           INC (Color);
55:           IF Color > ASH (1, Win^.wScreen^.bitMap.depth)-1 THEN
56:             Color := 0;
57:           END; (* IF *)
58:           g.SetAPen (Win^.rPort, Color); (* Farbe einstellen *)
59:         ELSE
60:         END; (* CASE *)
61:       ELSIF (I.mouseMove IN Class) THEN
62:         IF ((OldX # X) OR (OldY # Y)) AND Draw THEN
63:           OldX := X; OldY := Y; (* Um nicht 1 Pixel lange
                                       Linien zu zeichnen *)
64:           g.Draw (Win^.rPort, X, Y);
65:         END; (* IF *)
66:       ELSIF (I.closeWindow IN Class) THEN
67:         Quit := TRUE; (* und tschüß! *)
68:       END; (* IF *)
69:     END; (* LOOP *)
70:   UNTIL Quit;
71: CLOSE
72:   IF Win # NIL THEN I.CloseWindow (Win); Win := NIL END;
73: END MiniPaint.   © 1993 M&T
```

Listing 15: MiniPaint.mod zeigt die Betriebssystemprogrammierung mit Oberon

um den von diesem Element benötigten Speicher dem System zurückzugeben.

Wie schon erwähnt: in C ist die Pointer-Arithmetik nicht immer einfach, da sich schnell Fehler einschleichen, die der Compiler nicht moniert. Obwohl der Oberon-Compiler hier weitaus bessere Überprüfungen vornimmt, ist ein wenig Vorsicht angebracht. Es empfiehlt sich daher das Importieren des »NoGuru«-Moduls, um eventuelle Laufzeitfehler und deren Art erkennen zu können.

Module

Auch wir sind in der Lage, eigene Module zu schreiben und diese von anderen importieren zu lassen. Der grundlegende Aufbau eines Moduls ist bekannt. Was wir Ihnen bislang vorenthalten haben, ist die Möglichkeit, Bezeichner zu exportieren und der »CLOSE«-Teil.

Wir wissen, daß beim Aufruf eines Moduls der BEGIN-Teil ausgeführt wird. Der CLOSE-Teil wird vollzogen, wenn das Modul endet. Der große Vorteil von CLOSE liegt darin, daß dieser Teil auch dann abgearbeitet wird, wenn ein Laufzeitfehler auftritt oder der Abbruch mit <Ctrl C> (hierzu ist das Modul »Break« zu importieren) erzwungen wird. Das ermöglicht es, auch im Notfall noch alle zuvor angelegten Ressourcen freizugeben.

Nicht weniger bedeutungsvoll aber ist es, die im Modul existierenden Prozeduren, Variablen usw. für andere Module sicht- und nutzbar zu machen. Möchten Sie beispielsweise ein Programm schreiben, das IFF-Bilder anzeigt, wäre es praktisch, dieses Projekt in zwei Module zu unterteilen. Ein Modul kümmert sich um die Parameter, die das Programm übers CLI/Shell oder die Workbench erhält, das andere ist auf die Bearbeitung der IFF-Bilder spezialisiert und stellt Prozeduren zum Laden oder Speichern zur Verfügung.

Besonderes Augenmerk gilt den Modulen

Vorteil einer solchen Einteilung ist, daß das IFF-Modul auch von anderen Programmen zu verwenden ist und die Routinen nicht ständig neu programmiert werden müssen.

Möglich wird das durch die Exportmarkierung »*«. Mit ihr lassen sich globale Prozeduren, Variablen etc. exportieren. So können fremde Module auf diese zurückgreifen. Das fremde Modul muß lediglich das »Service«-Modul importieren. Beim Importieren wird der BEGIN-Teil des Moduls ausgeführt. Ist das Hauptmodul beendet, ruft Oberon die CLOSE-Teile der importierten Module in umgekehrter Reihenfolge auf. Jedes Modul kann für sich somit definierte Ausgangsbedingungen schaffen und bei Beendigung eventuell angeforderte Ressourcen zurückgeben. Bei Bedarf eröffnet das io-Modul im BEGIN-Teil beispielsweise ein Fenster, das im CLOSE-Teil geschlossen wird.

Als Beispiel zur Verwendung der Exportmarkierung dient wiederum das io-Modul. Betrachten Sie sich hierzu den auf der Compiler-Diskette befindlichen Quelltext. »*« folgt immer dem Namen einer Prozedur, Variablen usw.:

```
MODULE MeinModul;
TYPE
  Zeile*: ARRAY 80 OF CHAR;
VAR AllesKlar*: BOOLEAN;
  Intern: INTEGER;
  Text: Zeile;
PROCEDURE Hallo* (a: SET);
BEGIN
    (* ... *)
END Hallo;
BEGIN
  AllesKlar := TRUE;
CLOSE
  AllesKlar := FALSE;
END MeinModul.
```

Der Typ »Zeile«, die Variable »AllesKlar« und die Prozedur »Hallo« ist somit von anderen Modulen ansprechbar. In fremden Modulen schreibt man:

```
IMPORT mm: MeinModul;
```

Anschließend kann man wie gewohnt »mm.Zeile« als Typ verwenden oder die Prozedur »mm.Hallo« aufrufen.

Eine Besonderheit sind Records. Hier lassen sich einzelne Elemente exportieren, andere wiederum nicht. So gesehen ist es möglich, fremde Module an einem Zugriff auf Elemente, die sie nichts angehen, zu hindern (information hiding). Ein Beispiel:

```
VAR Demo*: RECORD
    text*: ARRAY 10 OF CHAR;
    intern: INTEGER;
    aha*: SET;
  END;
```

Module, die den Record »Demo« importieren, können nur auf die Elemente »text« und »aha« zugreifen. »intern« ist tabu.

Beim Erstellen eigener Module ist unbedingt darauf zu achten, sie voneinander weitgehend unabhängig zu gestalten: Jedes Modul sollte auch in einer anderen Umgebung einwandfrei laufen. Mit diesem Vorsatz läßt sich viel Arbeit bei der Fehlersuche und Erweiterung bestehender Module einsparen. Es ist zu vermeiden, globale Variablen fremder Module zu modifizieren bzw. von diesen abhängig zu sein.

Weitere Oberon-Module

Bis jetzt haben wir hauptsächlich das Modul io verwendet. Ein weiteres, nicht weniger wichtiges ist »Break«. Wird es importiert, ist es möglich, das Programm mit <Ctrl C> abzubrechen, und zwar beim Aufruf von Prozeduren mit eingeschalteter Stack-Überprüfung. Soll innerhalb einer Schleife ohne Prozeduraufrufe auf Abbruch getestet werden, ist die Funktion »CheckBreak« innerhalb der Schleife anzuspringen.

Wichtig sind zudem die beiden »NoGuru«-Module. Importiert man sie, bricht das Programm bei einem Laufzeitfehler nicht nur ab, sondern gibt zudem die Fehlerursache aus.

Um die Parameterübergabe an Programme zu vereinfachen, behilft man sich mit dem Modul »Arguments«, dessen Benutzung im Listing 13, »ArgDemo.mod«, demonstriert wird. Das Programm gibt alle Argumente numeriert auf dem Bildschirm aus. Starten Sie »ArgDemo« sowohl vom CLI/Shell als auch von der Workbench. »NumArgs« informiert über die Anzahl angegebener Argumente. Sind sie Null, hat man im CLI/Shell entweder keine Argumente angegeben, oder das Programm von der Workbench ohne weitere Icons aufgerufen. »GetArg« liefert das gewünschte Argument. Übergibt man als Parameter Null, erfährt man den Namen des Programms. Mit »GetLock« erhält man einen Lock auf das Verzeichnis.

Auch »FileSystem« ist ein zentrales Modul, worüber sich die Dateiverwaltung abwickeln läßt. »China.mod« (Listing 14) verwendet die Module »Arguments«, »FileSystem« und »io«.

Spielen wir jetzt eine der Stärken von Amiga-Oberon aus: Die Programmierung des Amiga-Betriebssystems. Ein einfacher, aber oft benötigter Betriebssystemaufruf ist der zum Öffnen eines Fensters. Sicherlich wissen auch Sie, daß das Amiga-Betriebssystem in Libraries unterteilt ist: Jede ist für einen bestimmten Aufgabenbereich zuständig.

Um ein Window (Fenster) zu öffnen, benötigt man die »intuition.library«. Um das Öffnen und Schließen müssen wir uns im Unterschied zu einigen anderen Sprachen nicht kümmern. Dies geschieht automatisch im BEGIN-Teil des Interface-Moduls »Intuition.mod«. Im CLOSE-Teil wird die Library selbstverständlich wieder geschlossen. Man muß also nur das zugehörige Interface-Modul importieren, und schon kann man alle Funktionen der Library verwenden.

Beginnen wir mit der »OpenWindow«-Funktion. Betrachtet man sich das entsprechende Interface-Modul oder liest man in der Bibel des Amiga-Programmierers nach (den ROM Kernel Reference Manuals [4], kurz RKMs), benötigt OpenWindow() als Parameter einen Record vom Typ »NewWindow«. Dessen Elemente muß man vor dem OpenWindow-Aufruf mit sinnvollen Werten belegen. Die Funktion liefert einen Zeiger auf die Window-Struktur, sozusagen den Schlüssel. Die Bedeutung der jeweiligen Elemente ist dem Kasten zu entnehmen. Möchte man das Amiga-Betriebssystem programmieren, was zwangsläufig nicht ausbleiben wird, ist die Anschaffung der RKMs erforderlich.

Listing 15 (»MiniPaint.mod«) demonstriert das Öffnen eines einfachen Fensters. Die Belegung der NewWindow-Struktur erfolgt mit strukturierten Konstanten. Mit »OpenWindow« öffnen wir es schließlich. Die Prozedur »Assert« des »Requests«-Moduls stellt sicher, daß das Fenster auch wirklich offen ist. Ist der Zeiger »Win« gleich NIL (undefiniert), gab's irgendwo Probleme. Mit »SetDrMd« legen wir den Zeichenmodus fest.

NewWindow-Flags für Oberon

leftEdge, topEdge, width, height: INTEGER;
Diese Werte geben die Abmessungen des Fensters an. Es ist darauf zu achten, daß sich das ganze Window innerhalb des Screens befindet.

detailPen, blockPen: SHORTINT;
Zeichenfarben, die für Titelzeile, Rahmen und Gadgets verwendet werden. Sollte man auf 0 und 1 setzen.

idcmpFlags: LONGSET;
Diese Flags geben an, welche Ereignisse dem Programm, zu dem das Window gehört, gemeldet werden.
sizeVerify: Intuition möchte die Größe des Fensters ändern. Man sollte kritische Ausgaben hinter sich bringen und die Message schnellstens beantworten.
newSize: Die Größe des Fensters wurde verändert.
refreshWindow: Das Fenster muß restauriert werden, da es zwischenzeitlich überdeckt war.
mouseButton: Eine Maustaste wurde gedrückt. Welche, erfährt man durchs »code«-Feld der »IntuiMessage«.
mouseMove: Die Maus wurde bewegt. Die neuen Koordinaten findet man ebenfalls in der »IntuiMessage«.
gadgetDown: Ein Gadget wurde aktiviert. Einen Zeiger darauf findet man im Element »iAddress« der »IntuiMessage«.
gadgetUp: Ein Gadget wurde losgelassen.
reqSet: Ein Requester wurde im Fenster geöffnet.
menuPick: Ein Menüpunkt wurde ausgewählt.
closeWindow: Das Close-Gadget des Fensters wurde betätigt.
rawKey: Eine Taste wurde gedrückt oder losgelassen. Im code-Feld der Intuition-Message findet man Code der betätigten Taste. Sondertasten lassen sich so hervorragend abfragen.
reqVerify: Intuition möchte einen Requester öffnen.
reqClear: Ein Requester wurde geschlossen.
menuVerify: Intuition möchte ein Menü darstellen.
newPrefs: Die Preferences (Voreinstellungen) wurden modifiziert.
diskInserted, diskRemoved: Eine Diskette wurde eingelegt bzw. entfernt.
wbenchMessage: Dient der systeminternen Verwendung. Nicht benutzen!
activeWindow: Window wurde aktiviert bzw. deaktiviert.
deltaMove: Ähnlich wie »mouseMove«, allerdings bekommt man hier nicht absolute Koordinaten, sondern die Änderung im Vergleich zur vorigen Position.
vanillaKey: Im Unterschied zu »rawKey« findet man im code-Feld den ASCII-Wert der Taste.
intuiTicks: Ist das Fenster aktiv, erhält man sie alle 1/50 Sekunde.
idcmpUpdate: Existiert erst ab Betriebssystem-Version 2.0 und wird für Boopsi-Gadgets benötigt.
menuHelp: Help-Taste wurde während der Menüauswahl gedrückt. Dieses Flag gibt's ebenfalls erst ab Betriebssystem-Version 2.0.
changeWindow: Neu ab OS 2.0: Position oder Größe des Fensters wurde verändert.
lonelyMessage: Systemintern. Nicht verwenden!

flags: LONGSET;
Hier gibt man die Eigenschaften des Fensters an.
windowSizing: Das Fenster läßt sich in seinen Ausmaßen vom Anwender verändern. Die Größenveränderung wird gemeldet, wenn man das IDCMP-Flag »newSize« setzt.
windowDrag: Das Window kann verschoben werden.
windowDepth: Das Window hat ein Gadget (OS 2.0 oder höher) oder Gadgets (OS 1.3 oder niedriger), um es in den Vorder- bzw. Hintergrund zu klicken.
windowClose: Das Fenster verfügt über ein Schließsymbol in der linken oberen Ecke. Man sollte das IDCMP-Flag »closeWindow« setzen, damit man erfährt, wann das Schließsymbol betätigt wurde.
sizeBRight: Dieses Flag hat nur Sinn in Verbindung mit »windowSizing«. Der rechte Rahmen über dem Size-Gadget wird verbreitert.
sizeBBottom: Ähnlich wie »sizeBRight«, hier wird der untere Rand verwendet.
simpleRefresh: Wird das Fenster überlagert, ist der Inhalt zerstört. Ist simpleRefresh« gesetzt, restauriert Intuition das Fenster nicht, sondern man ist selbst dafür verantwortlich. Nützlich ist hierbei das zugehörige IDCMP-Flag.
superBitMap: Übergroße BitMap. Mehr dazu in den RKMs.
backDrop: Das Fenster liegt immer hinter allen anderen. Sollte man nur auf eigenen Screens verwenden. Sinnvoll nur in Verbindung mit »borderless« (s.u.).
reportMouse: Man möchte immer über die Mausposition informiert werden. Zusätzlich ist das IDCMP-Flag »mouseMove« zu setzen.
gimmeZeroZero: Bitte nicht verwenden, wenn's nicht unbedingt sein muß. Verhindert das Zeichnen in den Window-Rahmen. Das so durchgeführte Clipping sollte man selbst vornehmen, da GZZ-Windows extrem langsam sind.
borderless: Das Fenster besitzt keinen Rahmen. Nur auf eigenen Screens in Verbindung mit »backDrop« (siehe oben) sinnvoll.
activate: Nach dem Öffnen des Fensters wird es aktiv.
windowActive, inRequest, menuState: Diese Flags werden vom System verwaltet. Sie signalisieren den momentanen Zustand des Fensters.
rmbTrap: Beim Betätigen der rechten Maustaste wird keine Menüzeile dargestellt. Dieses Flag ist dann notwendig, wenn man mit »mouseButtons« die rechte Maustaste abfragen will. Sinnvoll auch dann, falls das Fenster über kein Menü verfügt.
noCareRefresh: Kümmert man sich selbst um den Refresh des Windows, wird man mit diesem Flag von der Pflicht enthoben, »BeginRefresh« und »EndRefresh« aufzurufen.
windowRefresh: Das Window wird gerade erneuert.
wbenchWindow: Das Fenster wird auf der Workbench geöffnet.
windowTicked: Internes Flag, nur ein »Tick« pro 1/50 Sekunde
nwExtended: Es handelt sich um eine erweiterte NewWindow-Struktur. Das Flag ist wichtig, wenn man unterm OS 2.0 mit Tag-Listen arbeitet.
visitor: Wird unter OS 2.0 von Intuition gesetzt und zeigt an, daß es sich um einen Besucher-Window auf einem fremden Screen handelt.
zoomed: Das Window ist gerade im alternativen Größenzustand.
hasZoom: Das Window besitzt ein Zoom-Gadget (nur unter OS 2.0 und höher). Das Flag setzt Intuition.

firstGadget: GadgetPtr;
Zeiger aufs erste eigene Gadget des Fensters. Ist kein eigenes Gadget vorhanden, sollte man hier NIL eintragen.

checkMark: ImagePtr;
Hier kann man den Menü-Häkchen ein besonderes Aussehen verleihen. Gewöhnlich trägt man hier NIL ein.

title: e.STRPTR;
Hier gibt man die Adresse des Fenstertitels an. Die Adresse eines Strings erhält man mit Hilfe der Funktion »ADR« des »SYSTEM«-Moduls.

screen: ScreenPtr;
Hier tragen wir den Screen-Pointer ein, auf dem das Fenster geöffnet werden soll. Ist es der Workbench-Schirm, genügt NIL.

bitMap: g.BitMapPtr;
Hat man ein Fenster mit übergroßen Ausmaßen, in dem gescrollt wird, trägt man hier den Zeiger auf die Bitmap ein. Sonst NIL.

minWidth, minHeight, maxWidth, maxHeight: INTEGER;
Falls sich ein Sizing-Gadget am Fenster befindet, ist es ratsam, hier die minimalen und maximalen Fensterausmaße anzugeben. Trägt man bei den Max-Werten -1 ein, darf das Fenster beliebig groß sein.

type: SET;
Typ des Bildschirms, auf dem das Fenster erscheinen soll. Normalerweise ist dies der Workbench-Screen (»SET {I.wbenchScreen}«).

Der nun folgende Programmteil ist eine Schleife, die erst dann beendet wird, wenn die boolesche Variable »Quit« wahr ist. Mit »WaitPort« warten wir darauf, daß Intuition eine Nachricht für uns hat. Diese lesen wir mit »LOOP« und »GetMsg« aus.

Die Prozedur »Cut« modifiziert die Mauskoordinaten insofern, daß sie innerhalb der Fensterbegrenzung liegen und den Rahmen nicht überschreiben. Mit der linken Maustaste kann man Linien zeichnen. Dazu benötigen wir die Funktionen »Move«, »Draw« und »WritePixel« aus der Graphics-Library (siehe Importliste). Ein Druck auf die rechte Maustaste wechselt die Farbe. Mit »ASH« berechnen wir die Farbanzahl des Bildschirms. Im Close-Teil wird das Fenster wieder geschlossen.

Nun wünschen wir Ihnen viel Spaß mit der Programmiersprache Oberon. *rz*

Literatur- und Softwarehinweise

[1] Grundlagen Programmiersprachen, AMIGA-Magazin 11/92, Markt & Technik Verlag AG, Seite 166 ff.
[2] Amiga-Oberon-Compiler, A+L AG, Däderiz 61, CH-2540 Grenchen, Tel. 00 41 65/52 03 11, Fax 00 41 65/52 03 79
[3] Demo-Version des Oberon-Compilers, AMOK-PD-Disk 53
[4] Commodore Amiga, AMIGA Rom Kernel Reference Manual Devices, Third Edition, Reading 1991

Doppel-Plus

Amiga-C++

Auf der Kölner Amiga-Messe '92 gab's eine neue Programmiersprache zu bestaunen: »Maxon C++«. Was verbirgt sich hinter C++, was bringt es und wer braucht es? Wo liegen die Unterschiede zu Ansi-C?

von Jens Gelhar

Die Programmiersprache C ist vielen geläufig. Was bedeuten aber die zwei »++«? Bei C++ handelt es sich um das weiterentwickelte Ansi-C. Es besitzt die hervorstechende Eigenschaft, objektorientierte Programmierung zu unterstützen (OOP). Der Begriff ist z.Zt. in aller Munde, dennoch ist er erklärungsbedürftig.

Der objektorientierte Ansatz

Nahezu alle in der Praxis anzutreffenden Hochsprachen sind prozedural: Ein Programm ist in erster Linie eine Aneinanderreihung von Anweisungen. Daneben gibt's funktionale Programmiersprachen, deren grundlegendes Konzept das der Funktion ist (z.B. LISP). Eine weitere Spezies sind die in der Praxis wenig verbreiteten logischen Programmiersprachen (z.B. PROLOG), bei denen ein Programm die Beschreibung des Problems in Form logischer Ausdrücke darstellt. Näheres hierzu und zu Programmiersprachen im allgemeinen finden Sie in [1].

Der objektorientierte Ansatz ist ein neues Konzept, das die Daten eines Programms, also die eigentlichen »Objekte«, in den Vordergrund stellt. Der Programmierer ist also nicht mehr primär mit der für das prozedurale Konzept typischen Frage »Was soll mein Programm auf welche Weise machen?« konfrontiert, sondern überlegt zunächst, welche Datenobjekte das Programm benötigt und welche Eigenschaften diese haben.

Letzten Endes sind diese Objekte nichts anderes als Datenstrukturen, also das, was man in Pascal als »Record« oder in C als »struct« bezeichnet. Diesbezüglich hat also ein Objekt auch so etwas wie einen Datentypen. Da neben einer Ansammlung von Daten allerdings noch mehr gehört, redet man hier nicht von »Typen«, sondern von »Klassen«. In einer Klasse findet man zusätzliche Funktionen, die ihre Eigenschaften beschreiben.

Dazu ein fast schon klassisches Beispiel: Möchte man eine verkettete Datenliste implementieren, definiert man sich in einer prozeduralen Programmiersprache einen Datentypen, den man z.B. »Liste« nennt. Dazu gehören natürlich auch spezielle Funktionen, z.B. »Insert« zum Einfügen eines neuen Listenelements. Ein typischer Aufruf:

```
Insert(Liste, Neues Element).
```

In einer objektorientierten Programmiersprache – neben C++ sind »Simula«, »Smalltalk« und »Objective C« prominente Mitglieder dieser Familie – wird das Ganze differenziert gehandhabt. Hier wird man die Funktion »Insert« als eine Eigenschaft der Klasse Liste deklarieren, und man ruft sie z.B. mit

```
Liste.Insert(Neues Element)
```

auf. Verfechter der reinen Lehre objektorientierter Programmierung bezeichnen das dann nicht mehr als einen Funktionsaufruf, sondern sagen, hier werde die Methode »Insert« aufs Objekt »Liste« angewandt. Einfacher ist folgende Interpretation: Egal, was sich hinter dem Objekt »Liste« verbirgt – das Objekt hat die Operation Insert() auszuführen. In Smalltalk würde man sogar davon sprechen, daß an »Liste« die Nachricht »Insert (Neues Element)« geschickt wurde.

Auf den ersten Blick ist das zugegebenermaßen ein mäßig spektakulärer Unterschied in der Schreibweise, verbunden mit einer leicht esoterisch-verquast anmutenden Art, einen Programmtext zu deuten, zumal man die Insert-Funktion immer noch selbst schreiben muß und die so viel anders nicht aussieht als die einer prozeduralen Sprache. Interessant wird es aber, wenn die Programmiersprache mächtige Features fürs Definieren, Kombinieren und Modifizieren von Klassen zur Verfügung stellt. Der wichtigste Begriff in diesem Zusammenhang ist das Konzept der Vererbung.

Die Grundidee ist, von einer vorhandenen Basisklasse eine neue Klasse abzuleiten, indem man unter einem neuen Klassennamen Eigenschaften der Basisklasse modifiziert und vor allem neue Methoden und Daten hinzufügt. Diese Klasse »erbt« dann die Eigenschaften ihrer Basisklasse, sofern sie diese nicht verändert. In vielen Programmiersprachen kann man eine Klasse sogar von mehreren Basen ableiten – so lassen sich Eigenschaften von Klassen zusammenfassen.

Zurück zum Beispiel: Insert implementiert man objektorientiert eben nicht für eine ganz bestimmte Art von Listen (also nur eine Klasse), sondern zunächst für ganz allgemeine Listenstrukturen, von der sich letztlich nach Belieben ableiten läßt – z.B. eine sortierte Liste von Zeichenketten, aus der man wiederum durch Vererbung eine Adressenliste oder ein Telefonbuch kreiert.

Wer sich mit dem Amiga-Betriebssystem näher befaßt hat, dem dürften solche Konzepte bekannt vorkommen: Ein zentrales Element von »Exec« sind die Datentypen »List« und »Node«. Sie finden überall dort Anwendung, wo Listen benötigt werden: Für Task-, Libraries- und Message-Listen beispielsweise. Definiert sind sie in C und entsprechend umständlich und unflexibel ist das Ergebnis.

Vererbung ist ein Merkmal objektorientierter Sprachen

Zur objektorientierten Programmierung gehört aber mehr. Ein Objekt muß für gewöhnlich initialisiert werden. Hört es auf zu existieren, ist das »Aufräumen« fällig, also z.B. Ressourcen oder Speicher freigegeben. Durch »Konstruktoren« und »Destruktoren« wird das weitgehend automatisiert, so daß der Programmierer hier keinen weiteren Denkaufwand investieren muß. Die Daten entwickeln also scheinbar ein gewisses Eigenleben, was man auch als »lokale Intelligenz« bezeichnet.

Ferner umfassen objektorientierte Sprachen gewöhnlich noch Konzepte für die saubere Trennung von Implementation und Schnittstelle. Bei der Klasse »Liste« wäre z.B. die Funktion/Methode »Insert« von außen sicht- und von jedem Teil des Programms nutzbar. Implementatorische Details wie Pointer, mit denen sich Listenelemente verketten lassen, deklariert man vorzugsweise als private Daten der Klasse: So sind sie ausschließlich für die Funktionen der Klasse selbst sichtbar und lassen sich nicht von anderen Programmteilen manipulieren. Objektorientierte Sprachen wie beispielsweise C++ bieten Konzepte, solche Verbote penibel zu überwachen.

Dieser Artikel kann die Vorteile der objektorientierten Programmierung nicht umfassend darstellen oder gar eine Einführung in gebräuchliche Konzepte der neuen Art zur Programmierung bieten. Die primären Vorteile zeichnen sich allerdings schon aus den ersten Erklärungen ab:

Maxon C++

Maxon C++ gibt's in zwei Versionen. Die einfache Version setzt sich aus einem Ansi-C und C++-Compiler zusammen (in einer integrierten und einer CLI-Version). Enthalten ist weiterhin der Programmeditor »Edward«, der Oberflächengenerator »MakeAPP« und das Online-Hilfe-System »HotHelp«. Der Preis: Inklusive einem ca. 800seitigen deutschsprachigen Handbuch ca. 400 Mark.

Die Maxon C++-Developer-Version ist um den Source-Level-Debugger und den Macro-Assembler erweitert. Preis: ca. 600 Mark.

Bezugsadresse:
Maxon Computer GmbH, Schwalbacher Str. 52, W-6236 Eschborn/Ts., Tel. (0 61 96) 48 18 11

❐ Eines der wichtigsten Merkmale ist die gesteigerte Übersichtlichkeit des Quelltexts insofern, daß man zusammengehörende Daten und Funktionen auch gemeinsam deklariert und eindeutig festlegt, in welchem Programmteil auf bestimmte Bezeichner zugegriffen werden darf. Vor allem letzteres verbessert die Sicherheit und speziell die Wartbarkeit der Programme.

```
 1: #include <exec/types.h>
 2: #include <dos/dos.h>
 3: #include <clib/exec_protos.h>
 4: #include <rct/rctdef.h>
 5: #include <clib/rct_protos.h>
 6: #include <rct/rcttagfuncs.h>
 7: #include <iostream.h>
 8:
 9: APTR RctBase, app;
10:
11: void main() {
12:
13: if ((RctBase = OpenLibrary("rct.library",16)) &&
14:   (app = R_InitApplTags(TAG_DONE)) {
15:     if( R_FormAlertTags( app, RFA_DefaultID, 1L, RFA_AlertText,
16:     "[Wählen Sie Ihre Programmiersprache!][C++|ANSI-C]", TAG_DONE))
17:         cout << "Warum nur ANSI-C?";
18:     else
19:         cout << "Ihre Entscheidung ist goldrichtig!"
20:     R_ExitAppl(app);
21:     CloseLibrary((Library*)RctBase);
22:   }
23: }           © 1993 M&T
```

Listing 1: Ein mit Fehlern gespicktes Programm und Bewährungsprobe für den Compiler

❐ Ein weiterer Vorteil ist die Wiederverwertbarkeit von Quelltexten. Da es die Programmiersprache durch mächtige Konzepte unterstützt, komplexe Probleme erst ganz allgemein zu lösen und diese Teillösung dann für konkrete Anwendungen zu spezialisieren, muß man das Rad nicht immer wieder neu erfinden. Ganze Klassen aus anderen Projekten lassen sich leicht in ein neues Programm übernehmen. Tut man das nicht nur im stillen Kämmerlein sondern macht die Früchte seiner Arbeit auch anderen Programmierern zugänglich, bezeichnet man das als eine »Klassenbibliothek«.

Diese Vorteile machen sich insbesondere bei umfangreichen Projekten bemerkbar. Das ist unter anderem einer der Gründe, warum objektorientierte Programmiersprachen – allen voran C++ – im professionellen Bereich immer größere Verbreitung finden.

OOP und C++

Im Leben wird einem bekanntlich wenig geschenkt, und so muß man auch für die oben genannten Features bezahlen, vor allem dann, wenn man die »reine Lehre« des objektorientierten Paradigmas verwirklicht sehen will. Ein Aufruf einer Methode ist kein simpler Funktionsaufruf, denn es muß stets – und zwar meist zur Laufzeit des Programms – festgestellt werden, zu welcher Klasse ein Objekt gehört und was genau deshalb zu tun ist. Klar, daß das zu einem gewissen Overhead führt, sowohl in der Geschwindigkeit des Programms als auch im Speicherbedarf der Datenobjekte.

Das zündete schließlich die Idee, von dieser puristischen Form objektorientierter Programmiersprachen ein wenig abzurücken und dafür weniger schöne Programme in Kauf zu nehmen – mit dem Erfolg, die von prozeduralen Programmiersprachen gewöhnte Effizienz zu erzielen. Außerdem steht man in der rauhen Wirklichkeit außerhalb des akademischen Elfenbeinturms vor dem knallharten Fakt, daß Millionen Programmierer nicht umlernen und dann vielleicht auch noch Unmengen vorhandener Quelltexte in eine neue Sprache umschreiben wollen – eine ökonomische Tatsache, an der schon manche interessante neue Programmiersprache gescheitert ist. Es entstand also das Bedürfnis nach einer Sprache, die von einem etablierten Sprachstandard ausgeht und sie um objektorientierte Konzepte erweitert, ohne dabei den ursprünglichen Standard gravierend zu ändern.

Das Ergebnis war das auf Ansi-C aufgesetzte C++. Es stellt einen gelungenen Kompromiß aus der Schönheit objektorientierter Konzepte und der Kraft prozeduraler Sprachen dar. Da nur wenige Veränderungen am C-Kern von C++ vorgenommen werden, lassen sich C-Programme nach meist nur geringen Modifikationen von einem C++-Compiler übersetzen. Zudem bieten C++-Übersetzer oft Optionen, solche Unstimmigkeiten toleranter zu behandeln, oder sie lassen sich gleich in den Ansi-C-Modus umschalten.

Für den aufstiegswilligen C-Programmierer ist C++ also die erste Wahl, da er keine neue Sprache lernen muß, sondern Schritt für Schritt die zusätzlichen Features von C++ kennenlernen und ausprobieren kann.

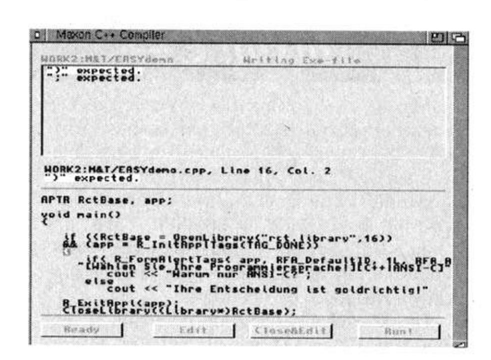


Bild 1: Editor und Compiler arbeiten in der Amiga-C++-Implementation Hand in Hand und ermöglichen so extrem kurze Entwicklungszeiten

Goodies und Features

Beim Entwickeln von C++ hat man nicht nur objektorientierte Konzepte zum guten alten C hinzugefügt, sondern auch noch eine Vielzahl interessanter und nützlicher Features aufgenommen, die zudem einiges von dem entschärfen, was C nach landläufiger Meinung so berüchtigt gemacht hat.

Das fängt bei Kleinigkeiten wie einer alternativen Schreibweise für Kommentare an (sie beginnen mit »//« und enden am Zeilenende, was gegenüber der C-Schreibweise mit »/* ... */« viel Tipperei spart) und hört bei den Operatoren »new« und »delete«, die die Verwaltung dynamischer Datenstrukturen vereinfachen (nie wieder »malloc«) noch lange nicht auf: Die etwas restriktivere Typprüfung und auch sonst etwas strenger gefaßte Sprachdefinition wird zwar einige hartgesottene C-Coder stören, erspart aber gerade dem Einsteiger etliche der Fallgruben, die C dem Programmierer gräbt. Referenzen machen die Arbeit mit Pointern leichter und ermöglichen endlich Funktionsparameter mit »Call by Reference«.

Eines der mächtigsten neuen Features aber ist die Möglichkeit der Überladung. Zum einen darf man Funktionen, vorzugsweise natürlich dann, wenn sie ähnliches tun, identische Namen geben, sofern sich die Parameterlisten unterscheiden. Der Compiler erkennt dann beim Aufruf einer solchen überladenen Funktion anhand der Argumente, welche Funktion gemeint ist. Betrachten wir uns einmal die Standard-Bibliotheken von Ansi-C: Dort findet man »abs«, »labs« und »fabs«, die alle dasselbe tun – nämlich den Betrag einer Zahl zu berechnen, und zwar für »int«, »long int« und »double«-Zahlen. In C++ ließen sich alle drei Funktionen »abs« nennen, was wesentlich ausdrucksstärker und intuitiver ist. Der Compiler erledigt den Rest.

Doch nicht nur Funktionen, auch Operatoren wie »+« oder »=« lassen sich überladen. Das eröffnet ganz neue Möglichkeiten: Hat man sich Klassen für Matrizen, Vektoren, komplexe Zahlen oder ähnliches definiert, kann man sich durch entsprechende Operator-Überladungen Datentypen schaffen, die den vordefinierten praktisch gleichgestellt sind: Schreibt man in C beispielsweise

```
MatrixMult(&a,b,c);
```

heißt es in C++ lediglich

```
a=b*c;
```

An Komfort und Lesbarkeit ist das wohl kaum zu überbieten. Viele Compiler bieten derartige Klassenbibliotheken im Lieferumfang, ebenso wie z.B. Klassen für dynamisch verwaltete Strings, die beliebig lang werden dürfen – einfach mit »+« aneinandergehängt. Das nimmt dann auch der berühmt-berüchtigten String-Behandlung von C (die eigentlich gar nicht existiert) ihren Schrecken.

Compiler-Übersicht

Ein Blick über den Tellerrand genügt, und man wird feststellen, daß bisher noch relativ wenige C++-Systeme auf dem Markt sind. Ein Grund ist der hohe Aufwand, den die

Implementierung einer so umfangreichen Sprache erfordert. Immerhin hält das den bei anderen Programmiersprachen bekannten Wirrwarr von Standards (und Abweichungen von denselben) in Grenzen.

Die erste C++-Implementierung überhaupt (nicht nur auf dem Amiga) ist der »Frontend« von AT&T. Frontend bedeutet, daß dieser Compiler die C++-Quelltexte nicht in ausführbaren oder assemblierbaren Code umwandelt, sondern einen C-Quelltext erzeugt, der dann von einem »anspruchslosen« C-Compiler endgültig übersetzt wird. Der Vorteil eines solchen Frontends (der selbst in C bzw. C++ geschrieben ist) liegt auf der Hand: Auf jedem System, für das es einen C-Compiler gibt, kann dieser Frontend unverändert benutzt werden. Der Nachteil: Eine solch zweistufige Übersetzung kann ganz schön lange dauern.

In Ermangelung eines verbindlichen Standards – die entsprechende Ansi-Kommission tagt zwar schon seit einiger Zeit, ist sich aber noch nicht so ganz einig – stellt diese Ur-Implementierung den De-facto-Standard dar. Seit etwa einem Jahr ist die Version 3.0 des Frontends verfügbar, weshalb man auch vom 3.0-Standard spricht. Davor gab es 2.0 und 2.1, die sich aber im wesentlichen durch Behebung von Fehlern (Bugs) und nicht in bezug auf den eigentlichen Sprachstandard unterscheiden.

Aus dem Unix-Bereich stammt »Gcc«, ein kombinierter Compiler für Ansi-C, C++ 3.0 und »Objective C«, der im Rahmen des GNU-Projekts entstand und somit als Public Domain, ja sogar als Quelltext verfügbar ist. Das System wurde inzwischen auf diverse Rechnerfamilien – einschließlich dem Amiga – portiert. Leider ist es ein ganz typisches Unix-System: Ein in jeder Hinsicht exzellenter und professioneller Compiler, der aber enorme Anforderungen an Speicherplatz, Festplattenkapazität und vor allem Geschwindigkeit stellt. Außerdem gibt es, wie man es von PD-Software kennt, kein richtiges Handbuch, also kein gedrucktes und schon gar kein deutsches. Mit dem Support sieht es naturgemäß ebenfalls nicht rosig aus. Auch der Debugger ist leider noch nicht portiert worden. Wer dennoch all das nicht scheut und einen entsprechend leistungsfähigen Computer (z.B. einen gut ausgestatteten Amiga 3000 oder gar einen Amiga 4000) besitzt, dem kann der Gcc nur wärmstens empfohlen werden.

Auf MS-DOS-Rechnern dürfte derzeit »Borland C++« der beliebteste C++-Übersetzer sein. Dieses System implementiert den 2.0-Standard mit einem zusätzlichen 3.0-Feature (den sog. Templates) und wird mit einer (zumindest vom Umfang her) imposanten Dokumentation zu einem für PC-Verhältnisse ausgesprochen fairen Preis geliefert. Daneben gibt es für diese Rechnerfamilie noch Portierungen des AT&T-Frontends sowie des Gcc und seit jüngstem auch »Microsoft C++«.

Sie dürfte es aber wohl in erster Linie interessieren, wie es auf dem Amiga aussieht. Vor einiger Zeit gab es »Lattice C++«, eine Portierung des guten alten Frontends (Version 1.x) in Verbindung mit der damals aktuellen Version von »Lattice C«. Es wurde aber aus schwer nachvollziehbaren Gründen nicht weiterentwickelt und ist inzwischen praktisch vom Markt verschwunden.

Bild 2: Der Amiga-C++-Compiler und der Source-Level-Debugger in voller Aktion – Programmiererherz, was willst du mehr?

Neben dem bereits erwähnten als PD erhältlichen Gcc sowie »Comeau C++« – einem Frontend ohne C-Compiler – ist das seit neuestem verfügbare MaxonC++ also die einzige Implementierung auf dem Amiga. Der Compiler hält sich an den 2.1-Standard und ist auch ohne größere Hardware-Anforderungen lauffähig. 1,5 MByte Speicher und eine Festplatte ist aber unbedingt empfehlenswert. Bei der Entwicklung wurde insbesondere auf Komfort Wert gelegt. Eine integrierte Entwicklungsumgebung, ein C++-Source-Level-Debugger, ein Intuition-Oberflächengenerator und ein Macro-Assembler sind Bestandteil des Compilers.

AT&T schuf mit »Frontend« den De-facto-Standard

Einer der wichtigsten Schalter ist der Umschalter von Ansi-C zu C++. Wie schon erwähnt, besteht so die Möglichkeit, Ansi-C-Quelltext einfach zu portieren und in kleinen Schritten der Objektorientiertheit anzupassen.

Anhand eines Beispielprogramms soll das Zusammenspiel zwischen Editor und Compiler aufgezeigt werden, wenn im Programmtext Fehler enthalten sind. Sie lassen sich mit der Demo-Version des Maxon-C++-Compilers übersetzen, die Sie auf der Diskette zum Heft finden (Seite 114). Starten Sie den Compiler nach Abtippen von Listing 1. Er wird die in Bild 1 deutlich zu erkennenden Fehlermeldungen ausgeben. Das Fehlerausgabefenster des Compilers zeigt alle während des Compilerlaufs auftretenden Bugs und die entsprechenden Quelltextzeilen an. Um die Fehler im Quelltext zu lokalisieren, muß die entsprechende Fehlerzeile im Compiler-Fenster angewählt und mit Anwählen von »Edit« an den Editor weitergeleitet werden. Der Cursor wird dabei automatisch auf der Fehlerstelle positioniert. Nach der Berichtigung der beiden Fehler und erneutem Compilieren kann das Programm direkt gestartet werden.

Fazit

Dem objektorientierten Ansatz wird sicher zu Recht nachgesagt, er werde die Software-Entwicklung revolutionieren. Bislang deutet alles darauf hin, daß C++ sich zu einer der wichtigsten Programmiersprachen der 90er Jahre entwickeln wird. Es kann jedem Programmierer nur empfohlen werden, sich mit dieser richtungweisenden Konzeption näher vertraut zu machen.

Abschließend möchten wir Ihnen noch ein vielsagendes C++-Programm vorstellen – eine rudimentäre Implementierung komplexer Zahlen. In manchen Programmiersprachen (z.B. Fortran) kann man direkt mit komplexen Zahlen rechnen. C++ kennt hingegen keine oder ähnliche mathematische Features. Das muß kein Nachteil sein, denn sie sind bei Bedarf leicht selbst zu implementieren. Listing 2 stellt eine solche dar.

Es gliedert sich in drei Teile: eine Schnittstellendefinition mit den Deklarationen der Klasse »Complex« sowie der übrigen Operator-Funktionen, ein Implementationsteil und ein Beispielprogramm.

Die wichtigste Deklaration ist die der Klasse Complex. Das Schlüsselwort »public« gibt an, daß alle nachfolgenden Bezeichner von jedem Teil des Programms benutzt werden dürfen. Es gibt hier also keine privaten Daten, auf die nur aus der Klasse selbst zugegriffen werden darf. In der nachfolgenden Zeile werden die Member »r« und »i« für den Real- bzw. Imaginärteil der Zahl deklariert. Das entspricht Struktur-Membern in C oder Record-Feldbezeichnern in Pascal. Interessanter ist die folgende Deklaration einer Funktion, die wir mit dem gleichen Namen wie die Klasse (nämlich »Complex«) bezeichnen und zwei Parameter besitzt. Zum einen dem Konstruktor, der Objekte des Typs Complex automatisch initialisiert. Bei der Deklaration eines solchen Objekts läßt sich demzufolge wahlweise Real- und Imaginärteil angeben, z.B.:

`Complex a(1,0), b(-0.5,1.2);`

oder einen oder beide Werte weglassen:

`Complex c(42), d;`

Das ist deshalb möglich, weil hier für die Parameter Default-Argumente (»= 0«) angegeben sind. Der Compiler setzt solche Argumente automatisch ein, wenn bei einem Konstruktor- oder Funktionsaufruf Argumente fehlen.

Den Abschluß der Klassendefinition bildet der Prototyp der Member-Funktion (oder auch Methode) »conj«. Diese Funktion läßt sich auf Objekten des Typs Complex aufrufen und liefert die konjugiert-Komplexe. Das angehängte »const« teilt dem Compiler mit, daß »conj« das Objekt, auf dem es aufgerufen wird, nicht verändert.

Nach der Klassendefinition folgen die zu überladenden Operatoren. Dies sind »+«, »-«, »*« und »/« für die vier Grundrechenarten sowie die Vergleichsoperationen »==« und »!=«. Da es auf komplexen Zahlen bekanntlich keine vollständige Ordnung gibt, können wir uns »kleiner«, »größer« usw. ersparen. Das »&« vor den Parametern gibt an, daß die Argumente als Referenz zu übergeben sind.

Es folgt noch eine interessante Operator-Definition, die ausführlicherer Erläuterung bedarf: Die Ausgabe von Daten erfolgt in C++ über die Klasse »ostream«, auf die der Operator »<<« überladen ist. Das Objekt »cout« der Klasse »ostream« steht z.B. für die Standard-Ausgabe. Mit Anweisungen wie

`cout << "Das Ergebnis ist " << x << ".";`

lassen sich Daten aller Art ausgeben. Der Programmierer selbst kann »<<« weiter überladen und so Ausgabefunktionen für eigene Datentypen definieren – in diesem Fall als Routine für die Klasse »Complex«.

Die eigentliche Implementation der so definierten Funktionen und Operatoren birgt für C-Programmierer nicht viel neues, einmal abgesehen davon, daß bei »conj« der Klassenname angegeben werden muß (verschiedene Klassen können nämlich gleichnamige Funktionen besitzen) und daß ein Operator wie eine ganz normale Funktion aussieht, nur daß ihr Name eben z.B. »operator +« ist.

Das Listing demonstriert die Klasse »Complex«. Variablen (Objekte) der Klasse »Complex« werden wie vordefinierte numerische Variablen deklariert und lassen sich direkt mit den Operatoren verknüpfen. *rz*

Quell- und Literaturhinweise

[1] Von ARexx bis Pascal – Binäre Inflation, AMIGA-Magazin 11/92, Seite 172 ff.

[2] Zeitler, Rainer: Maxon C++-Compiler – Software »made in Germany«, AMIGA-Magazin 12/92, Seite 98

[3] Meyzis, Edgar: Klassenhaftes C, AMIGA-Magazin 1/93, Seite 176

```
  1: // Einfache Implementierung komplexer Zahlen in C++
  2: // Geschrieben von Jens Gelhar 20.11.92
  3: #include <iostream.h>     // Includedatei für Ein- und Ausgabe
  4:
  5: // ***** Die Klassendefinition und Funktionsprototypen *****
  6: class Complex {
  7:     public:                     // d. h. keine privaten Bezeichner
  8:
  9:     double r, i;                // Real- und Imaginärteil
 10:
 11:     Complex(double re=0, double im=0);  // Konstruktor
 12:
 13:     Complex conj() const;     // berechnet konjugiert-komplexe
 14: };
 15:
 16: // Die vier Grundrechenarten:
 17: Complex operator + (const Complex &c1, const Complex &c2);
 18: Complex operator - (const Complex &c1, const Complex &c2);
 19: Complex operator * (const Complex &c1, const Complex &c2);
 20: Complex operator / (const Complex &c1, const Complex &c2);
 21:
 22: // Vergleichs-Operationen:
 23: int operator == (const Complex &c1, const Complex &c2);
 24: int operator != (const Complex &c1, const Complex &c2);
 25:
 26: // Ausgabe-Funktion:
 27: ostream &operator << (ostream &os, const Complex &c);
 28:
 29: // ***** Implementation *****
 30: Complex::Complex(double re, double im) {
 31:   r = re;
 32:   i = im;
 33: }
 34:
 35: Complex Complex::conj() const {
 36:   Complex ergebnis(r, -i);
 37:   return ergebnis;
 38: }
 39:
 40: Complex operator + (const Complex &c1, const Complex &c2) {
 41:   Complex ergebnis(c1.r+c2.r, c1.i+c2.i);
 42:   return ergebnis;
 43: }
 44:
 45: Complex operator - (const Complex &c1, const Complex &c2) {
 46:   Complex ergebnis(c1.r-c2.r, c1.i-c2.i);
 47:   return ergebnis;
 48: }
 49:
 50: Complex operator * (const Complex &c1, const Complex &c2) {
 51:   Complex ergebnis;
 52:   ergebnis.r = c1.r*c2.r - c1.i*c2.i;
 53:   ergebnis.i = c1.r*c2.i + c1.i*c2.r;
 54:   return ergebnis;
 55: }
 56:
 57: Complex operator / (const Complex &c1, const Complex &c2) {
 58:   Complex nenner = c1 * c2.conj();
 59:   double zaehler = c2.r*c2.r + c2.i*c2.i;
 60:   return Complex(nenner.r/zaehler, nenner.i/zaehler);
 61: }
 62:
 63: int operator == (const Complex &c1, const Complex &c2) {
 64:   return (c1.r == c2.r && c1.i == c2.i);
 65: }
 66:
 67: int operator != (const Complex &c1, const Complex &c2) {
 68:   return (c1.r != c2.r || c1.i != c2.i);
 69: }
 70:
 71: ostream &operator << (ostream &os, const Complex &c) {
 72:   if(c.r != 0)
 73:   { if (c.i > 0)
 74:       os << "(" << c.r << " +" << c.i << "i)";
 75:     else if (c.i < 0)
 76:       os << "(" << c.r << " -" << -c.i << "i)";
 77:     else
 78:       os << c.r;
 79:   }
 80:   else
 81:    if (c.i != 0)
 82:      os << c.i << "i";
 83:    else
 84:      os << "0";
 85:   return os;
 86: }
 87:
 88: // ***** Ein simples Beispielprogramm: *****
 89: void main() {
 90:   Complex x, y;
 91:
 92:   cout << "Bitte geben Sie zwei komplexe Zahlen ein:" << endl;
 93:   cout << "x (Realteil):     ";
 94:     cin >> x.r;
 95:   cout << "x (Imaginärteil): ";
 96:     cin >> x.i;
 97:
 98:   cout << "y (Realteil):     ";
 99:     cin >> y.r;
100:   cout << "y (Imaginärteil): ";
101:     cin >> y.i;
102:
103:   cout << endl;
104:   cout << "x + y = " << x+y << endl;
105:   cout << "x - y = " << x-y << endl;
106:   cout << "x * y = " << x*y << endl;
107:   cout << "x / y = " << x/y << endl;
108: }               © 1993 M&T
```

Listing 2: Die Implementation Komplexer Zahlen ist in C++ schnell erledigt

Modula-2-Ableger

Nur der Amiga hat sie – Cluster

Cluster ist eine für den Amiga nicht mehr ganz neue Programmiersprache. Sie lehnt sich ans Modula-2-Konzept an, verfügt allerdings über einige außergewöhnliche Besonderheiten.

von Thomas Pfrengle

Auf dem Amiga gibt's jede Menge Programmiersprachen. Am beliebtesten ist »C« – nicht zuletzt aufgrund des Amiga-Betriebssystem-Konzepts. Mittlerweile mausern sich Oberon-2 und Modula-2 zu ernsten Rivalen des scheinbar allmächtigen C. Auch Cluster mischt seit zwei Jahren mit.

Cluster basiert auf Modula-2 und ist nach wie vor in der Lage (Version 2.0, siehe Kasten), Quelltexte dieser Sprache zu übersetzen. Das Sprachkonzept von Modula-2 hingegen schöpft nicht alle Cluster-Fähigkeiten aus und ist nur dann angebracht, wenn Portabilität unumgänglich ist. Wie sieht nun ein Cluster-Programm aus? Bild 1 demonstriert den grundsätzlichen Aufbau. Die Elemente IMPORT, TYPE, VAR, CONST, EXCEPTION und PROCEDURE dürfen in beliebiger Reihenfolge auftreten, sind aber nicht zwingend. Ebenso ist der CLOSE-Teil optional. In Implementationsmodulen ist der BEGIN-Teil ebenfalls unnötig.

Der Importteil

Auffällig ist der IMPORT-Teil, da er doch einige Unterschiede zu Modula-2 aufweist. Für all die Leser, die mit Modula nicht vertraut sind, einige Anmerkungen zum Modul-Konzept: Ein Programm besteht im allgemeinen nicht nur aus einem Modul. Dies hat zum einen den Vorteil, daß Programme überschaubar bleiben, zum anderen bietet das die Möglichkeit, Prozeduren und Typen eines speziellen Gebiets zusammenzufassen. So lassen sie sich einfach importieren, ohne eine Routine immer wieder neu zu schreiben. Da die einzelnen Module nur einmal übersetzt und anschließend hinzugelinkt werden, spart man bei der Übersetzung des Hauptmoduls zusätzlich Zeit.

Derartige Bibliotheksmodule haben einen geringfügig modifizierten Aufbau als der normaler Module: Sie setzen sich aus einem Definitions- und Implementations-Teil zusammen. Die Trennung hat gegenüber dem Konzept von Oberon (nur ein Modul, das sowohl Implementation als auch Definition enthält) folgende Vorteile: Einmal entspricht sie mehr dem Prinzip des Information-Hidings (Verbergen von Informationen), da bei Großprojekten mit vielen Programmierern nur die Definitionsmodule untereinander auszutauschen sind, ohne daß die Implementationsteile explizit bekannt sein müssen (Know-how-Schutz). Zum anderen verhindert man so, daß unbeabsichtigt Implementationsmodule verändert werden, was zu Inkonsistenzen im Gesamtprojekt führte (wenn jeder Programmierer unterschiedliche Implementationsteile besäße). Außerdem lassen sich so z.B. hardwarenahe Prozeduren kapseln und der Benutzer wird mit irgendwelchen Registern gar nicht erst konfrontiert – für sie ist es ein ganz normales Hochsprachenmodul.

Ein weiterer Vorteil der Trennung ist, daß bei einer Änderung in der Implementation nicht alle davon abhängigen Module neue zu kompilieren sind. Das gilt aber nur, solange die Definitionsdatei unverändert bleibt. Ein zusätzlicher Nebeneffekt ist die gute Übersichtlichkeit und Dokumentation der Schnittstelle. Man erkennt recht schnell, was sich



von einem Modul importieren läßt. In Oberon-2 z.B. kennzeichnet man zu exportierende Objekte mit dem Sternoperator »*«. Durch das Trennen in Definitions- und Implementationsmodul wird es möglich, daß sich zwei Module gegenseitig importieren können. Bei Cluster besteht darüber hinaus die Möglichkeit, Definitionsmodule gegenseitig zu importieren. Das ist z.B. fürs DOS notwendig, da in »Dos.Prozess« ein Window-Pointer aus dem Intuition-Modul benötigt wird, Intuition aber auch das Dos-Modul importiert.

Zurück zum IMPORT-Teil. Dort läßt sich all das importieren, was im Definitionsteil eines anderen Moduls definiert wurde. Das kann auf verschiedene Arten geschehen:

❒ Direkter Import: Es wird der Name des zu importierenden Objekts angegeben, z.B. »Read«. Daraufhin ist das Objekt direkt über seinen Namen ansprechbar, ohne es jedesmal mit dem Modulnamen zu qualifizieren.

❒ Qualifizierter Import: Man verwendet die einzelnen Objekte, indem man sie über den Modulnamen qualifiziert, z.B. »InOut.ReadString«. Dies ist dann notwendig, falls in zwei Modulen gleiche Bezeichner vorkommen und man sie in einem Modul verwendet. Um Tipparbeit zu sparen, bietet das Schlüsselwort »AS« die Möglichkeit, sie umzubenennen. AS ist optional, kann auch fehlen.

❒ Import über Importgruppen: Mit dem Schlüsselwort »GROUP« lassen sich in Definitionsmodulen Bezeichner oder andere Importgruppen zu einer Einheit zusammenfassen, z.B. »WriteGrp«. Somit entfällt die

Rund um Cluster

Cluster ist eine integrierte Entwicklungsumgebung: Aus dem Editor heraus läßt sich direkt in den Speicher kompilieren, linken, »maken« und das Programm starten. Beim Kompilieren hält Cluster die Symboldateien automatisch in einem dynamischen Cache – so verhindert man ständiges Nach- bzw. Neuladen schon verwendeter Symboldateien. Der Loader, verantwortlich fürs Ausführen von zuvor übersetzten Programmen, besitzt die Fähigkeit, auftretende Laufzeitfehler abzufangen und mit Zeilen- und Modulangabe im Quelltext zu melden.

Der Editor verfügt neben vielen anderen Funktionen wie automatischem Fettdruck verwendeter Schlüsselwörter über die Möglichkeit, alle Tasten mit Makros zu belegen. Die eigenwillige Oberfläche des Editors war in der Vergangenheit ein Kritikpunkt. Mit der neuen Cluster-Version V2.0 wird ein Amiga-konformer Editor sowie Cli/-Shell-Versionen von Compiler, Linker, Loader und Make mit AREXX-Ports ausgeliefert, ebenso eine Vorversion vom Debugger.

Beim Compiler handelt es sich um einen Single-Pass-Compiler (6000 Zeilen pro Minute Übersetzungsgeschwindigkeit und über 1600 Dhrystones/s auf einem MC68000-Amiga; bei einem Amiga mit 33 MHz MC68030-Prozessor erreicht er sogar 13 000 Dhrystones/s). Er generiert lauffähigen Code für alle 68xxx-Prozessoren einschließlich dem MC68040 und dem 68882-Koprozessor. Reentrante Programme im kleinen Datenmodell lassen sich genauso kreieren wie Libraries und Devices. Der selektiv operierende Linker bindet nur die wirklich benötigten Objekte und erstellt so sehr kompakte Programme.

Bezugsadresse: DTM GmbH, Dreiherrenstein 6a, W-6200 Wiesbaden-Auringen, Tel.: (0 61 27) 40 65, Fax (0 61 27) 6 62 76
Preis: ca. 400 Mark

lange Importliste am Modulanfang. Im übrigen ist nicht nur zu Beginn eines Moduls das Importieren erlaubt: auch innerhalb einer Prozedur geht das.

Die Typdeklaration

Die in Cluster verwendeten Standardtypen finden Sie in Bild 2. Der Typ »STRING« ist hier ein echter Stringtyp und nicht wie in C oder Modula als »ARRAY OF CHAR« mit abschließendem Null-Byte definiert. Statt dessen ist er als RECORD mit der Länge (Integer) an erster Position und einem Null-Byte-terminiertem »ARRAY OF CHAR« implementiert. Hierbei ist das letzte Null-Byte nicht in der Länge enthalten:

```
STRING  =  RECORD
       len  : INTEGER;
       data : ARRAY [0..MAX] OF CHAR
    END
```

Diese Definition bietet enorme Vorteile: Abhängig von der gestellten Aufgabe ist die Behandlung eines Strings unterschiedlich effizient: Möchte man z.B. den String kopieren, richtet man sich nach dem Null-Byte. Fürs Parsen bietet sich hingegen das Längenelement an. Cluster erlaubt sogar die Zuweisung von Strings mit unterschiedlicher Maximallänge. Hier prüft der Compiler, ob sie zulässig ist.

Neben vorgegebenen Standardtypen lassen sich eigene anlegen:

❍ Unterbereichstypen z.B. [3..211]

❍ Aufzählungstypen, z.B. (Rot,Gruen,Blau).

Cluster erlaubt die Angabe von Elementen innerhalb eines Aufzählungstypen mit gleichem Namen. Der Compiler versucht dann, aus dem Kontext den richtigen Typ zu ermitteln. Ist das nicht möglich, muß man das Element über den Typnamen qualifizieren. Außerdem lassen sich Aufzählungstypen mit einem vorgegebenen Wert starten oder nach einer Lücke mit einem anderen Wert fortsetzen. Somit vermeidet man lästige Fülleinträge in den Systemmodulen.

❍ Flags = (Flag1,Flag2,Flag3,Flag255=255,Flag256);

❍ Mengen: Aus Aufzählungs- oder Unterbereichstypen mit maximal 32 Elementen lassen sich Sets oder auch Mengen bilden. Im Gegensatz zu C und Oberon überprüft Cluster, ob die Flags zum Set-Typen passen. Es ist demnach unmöglich, ein Window-Flag und ein Gadget-Flag als Screen-Flags zu mißbrauchen. Bei der Wertzuweisung ist es nicht mehr notwendig, den Typnamen anzugeben:

```
FlagSet = SET OF [Flag1..Flag3];
SetVar := {Flag,Flag3}
```

❍ Records: Records sind Zusammenfassungen mehrerer Variablen. Sie sind für Daten, die man z.B. in einem Rutsch einer Prozedur übergeben oder speichern möchte, ideal. Auf die einzelnen Record-Elemente greift man zu, indem man sie über den Record-Elementnamen anspricht. Record-Elemente dürfen jeden Typ besitzen, sogar Records selbst. Von Records lassen sich Nachfolger definieren, die zuweisungskompatibel zu ihrem Vorgänger sind. Ein Beispiel zeigt Bild 3.

Das erste Element des Records »Message« ist kein Node, da es als Nachfolger von Node definiert ist. Dieses läßt sich demzufolge an alle Prozeduren übergeben, die eine Node-Struktur erwarten, ohne eine Pointer-Konvertierung durchführen zu müssen.

❍ Arrays: Man kann sie sich als eine Elementliste eines Typs vorstellen, die über einen Index anzusprechen sind. Sie eignen sich hervorragend im Zusammenhang mit Schleifen, möchte man auf viele Elemente die gleiche Operation anwenden.

Selbstverständlich sind auch mehrdimensionale Arrays möglich. Außerdem läßt sich nicht nur mit festen, sondern auch mit offenen Arrays arbeiten. Von diesem Typ kann man sich zwar keine Variablen anlegen, sie bieten jedoch die Option, vom offenen einen festen Typ abzuleiten, der dann zum offenen zuweisungskompatibel ist (daher eignen sich offene Arrays sehr gut als Übergabeparameter an Prozeduren). Zudem ist es möglich, einen Pointer-Typen zu definieren und anschließend ein Feld bestimmter Länge zu reservieren. Der Compiler nimmt auch hier eine Bereichsüberprüfung vor.

❍ Prozedurtypen: Einer Variablen dieses Typs lassen sich alle Prozeduren zuweisen, die dem Aufbau des Prozedurtyps entspricht:

```
TYPE ProcType = PROCEDURE(i : INTEGER;
                          name : STRING);
```

Diesem kann man sich all die Prozeduren zuweisen, deren erster Parameter ein INTEGER- und zweiter ein STRING-Wert ist. Wo ist der praktische Nutzen eines solchen Typs? Stellen Sie sich vor: Sie rufen an mehreren Stellen eine Prozedur auf. Unter bestimmten Umständen benötigt man an dieser Stelle aber eine andere Prozedur gleichen Typs. Der aufwendige Weg sieht so aus, an jeder Stelle eine IF-Abfrage zu implementieren und entsprechend die Funktion aufzurufen. Einfacher ist es, anstelle von Prozeduren eine Prozedurvariable einzuführen und diese vor dem Aufruf adäquat zu belegen.

```
MODULE BeispielModul
FROM InOut AS io IMPORT Read,WriteGrp;
IMPORT FileSystem AS fs;
TYPE
  Mögliche Typdeklarationen
VAR
  Mögliche Variablendeklarationen.
CONST
  Mögliche Konstantendeklarationen
EXCEPTION
  Mögliche Exceptiondeklarationen
PROCEDURE
  Procezedurdeklarationen
GROUP
  Deklaration von Importgruppen (Nur in
  Defintionsmodulen)
BEGIN
  Hauptprogramm
CLOSE
  Closeteil
END Beispielprogramm.    © 1993 M&T
```

Bild 1: So präsentiert sich das Grundgerüst eines Cluster-Programms – die Ähnlichkeit zur Programmiersprache Modula-2 ist unverkennbar

SHORTCARD, CARDINAL, LONGCARD
SHORTINT, INTEGER, LONGINT
FFP, REAL, LONGREAL
BOOLEAN, LONGBOOL (Long hier zur C-Kompatibilität in Systemmodulen)
CHAR, LONGCHAR
ANYTYPE: Kann jeder Typ sein; nur als Übergabeparameter zu gebrauchen; man kann nur auf seine Länge und seine Adresse zugreifen (vergleichbar zu 'ARRAY OF BYTE' in Modula-2).
ANYPTR: Vergleichbar ADDRESS in Modula-2 sowie STRING.

Bild 2: Die von Cluster zugelassenen Variablentypen – einige wurden in Anlehnung ans Amiga-Betriebssystem implementiert

In Cluster besteht hier nun die Möglichkeit (die sonst nur noch in Modula-3 vorhanden ist), an einen Übergabeparameter einer Prozedur nicht nur globale Prozeduren zu übergeben, sondern auch lokale. Oft ist das vorteilhaft, da lokale Prozeduren auch auf lokale Variablen der sie umgebenden Prozeduren zugreifen können.

❍ Konstanten : Konstanten werden in Cluster wie in Modula-2 deklariert, d.h.

```
CONST
    Name = Wert,
```

Dabei lassen sich nicht nur einfache Konstanten deklarieren, auch Konstanten beliebigen Typs, also Records, Arrays oder Pointer, sind davon nicht ausgenommen.

Selbst das Vordefinieren von Konstanten ist gültig. Gibt man z.B. im Definitionsmodul eine Konstante an, der letztlich aber erst im Hauptprogramm ein Wert zugewiesen wird, vermeidet man, daß bei einer Wertveränderung alle Module erneut zu kompilieren sind. Diese Fähigkeit ermöglicht es auch, konstante verkettete Listen aufzubauen. So deklariert man z.B. viele Systemstrukturen als Konstanten und muß die jeweiligen Werte nicht während der Laufzeit zuweisen.

Variablendeklarationen

Zum Teil kennen wir sie schon aus vorigen Beispielen. Da es sich bei Cluster um einen Single-Pass-Compiler handelt, ist es notwendig, Variablen vor der ersten Verwendung anzugeben – ansonsten meckert er. Das geschieht durch Angabe des Schlüsselworts »VAR« gefolgt von den Namen. Den Namen schließt sich der Doppelpunkt und eine Typdefinition an. Bis hier entspricht das genau der Syntax von Modula-2. In Cluster aber dürfen Sie die Variablen zusätzlich mit einem Initialwert versehen:

```
VAR i      : INTEGER :=10
  Feld  :=TestArray:(1,2,3,4,5,3,6,6,-3,0);
```

Möchte man, daß eine Variable ständig in einem Register liegt, kann man es bei der Deklaration angeben:

```
VAR  i  IN D2 : INTEGER;
```

Durch das Schlüsselwort »STATIC« erreicht man, daß lokale Variablen nach Verlassen der Prozedur ihren Wert nicht verlieren, sondern beim erneuten Aufruf immer noch denselben Wert innehaben. Im Prinzip ließe sich hierfür auch eine globale Variable verwenden. Der Vorteil von STATIC ist jedoch der, daß die Variable nur innerhalb der Prozedur bekannt ist.

❍ Attribute: Darunter versteht Cluster spezielle Eigenschaften von Typen und damit auch von Variablen, die durch »Variablen/Typnamen'Attributname« abzufragen sind (Bild 4).

❍ Der BEGIN-Teil: Hier findet der eigentliche Programmablauf statt. Er besteht aus Anweisungen, getrennt durch Semikola. Die von Cluster angebotenen Kontrollstrukturen sind vielfältig. Um Variablen mit mehreren Werten zu vergleichen, verwendet man in Modula-2 die »CASE«-Anweisung. In Cluster mutierte sie zwecks größerer Geschlossenheit (s. »WHILE«) zu einem »IF KEY«.

In Cluster darf innerhalb des CASE- bzw. IF KEY-Konstrukts der Vergleich nicht nur, wie in Modula-2 üblich, mit Konstanten erfolgen, auch Variablen sind erlaubt. Die Anweisungsfolge wird allerdings nur dann ausgeführt, wenn ebenfalls die AND_IF-Anweisung wahr ist. So entstand ein Nebeneffekt: Der »OF«-Operator ist für allgemeine Boolesche Bedingungen frei (z.B. in WHILE- oder REPEAT-Schleifen):

```
IF c OF "a","b","c".."e",var1 THEN END
```

```
MinNodePtr     = POINTER TO MinNode;
MinNode        = RECORD
                   succ,
                   pred  : SAMEPTR;
                 END;
NodePtr        = POINTER TO Node;
Node           = RECORD OF MinNode
                   type  : NodeType;
                   pri   : NodePri;
                   name  : SysStringPtr;
                 END;
MessagePtr     = POINTER TO Message;
Message        = RECORD OF Node
                   replyPort : MsgPortPtr;
                   msgSize   : CARDINAL;
                 END;
```

Bild 3: Records spielen eine zentrale Rolle im Sprachkonzept von Cluster und werden insofern extrem flexibel gehandhabt

Dabei wird c mit jedem der nachfolgenden Werte verglichen. Während des Vergleichs verharrt c jedoch in einem Register, was einen Geschwindigkeitsgewinn zur Folge hat.

Schleifen

❍ Die REPEAT-Schleife:

```
REPEAT
  Anweisungsfolge
UNTIL (Boolesche Bedingung);
```

❍ Die WHILE-Schleife:

```
WHILE (Boolesche Bedinung) DO
  Anweisungsfolge
END;
```

Die Schleife wird nur dann ausgeführt, wenn die Boolesche Bedingung wahr ist. Im Unterschied zur REPEAT-Schleife ist die Anweisungsfolge nicht unbedingt abzuarbeiten. WHILE wurde um »OR_WHILE« und »ELSE« erweitert. Durch diese Maßnahme ist die LOOP-Schleife praktisch überflüssig.

❍ WITH: Diese Anweisung erlaubt im allgemeinen den vereinfachten Zugriff auf RECORD-Elemente. Eine Einschränkung war bislang allerdings hinzunehmen: Der Zugriff ist nur auf einen Record desselben Typs gleichzeitig möglich. Bei Cluster gibt's diese Einschränkung nicht. WITH wurde insofern modifiziert, daß es sich auf jeden TYP anwenden läßt. Eine Variable einfachen Typs befördert Cluster über den gesamten WITH-Bereich in ein Register, was zu einem Geschwindigkeitsgewinn führt.

Prozeduren und Funktionen

Innerhalb von Prozeduren und Funktionen dürfen die gleichen Konstrukte auftauchen, die wir schon vom Modul kennen.

Cluster-Prozeduren und -Funktionen haben jedoch einige Besonderheiten: Als Übergabeparameter gibt es VAR-, REF-, und Werteparameter. Der Unterschied: Bei einem Werteparameter wird der Wert kopiert – Veränderungen in der Prozedur beziehen sich also nicht auf die übergebene Variable. Bei VAR-Parametern wird nur ein Zeiger auf die Variable übergeben, Änderungen in der Prozedur verändern auch die übergebene Variable außerhalb der Prozedur. In vielen Fällen möchte man das ausschließen. Handelt es sich aber z.B. um ein Array, ist kopieren zu zeitaufwendig. Hier verwendet man den REF-Parameter. Auch hier übergibt man nur die Adresse, Schreibzugriffe innerhalb der Prozedur sind nicht möglich.

Übergabeparameter lassen sich als Registerparameter definieren:

```
PROCEDURE Reg(i IN D1 : INTEGER);
```

Es ist möglich, in Cluster Funktionen mit komplexem Typ als Rückgabewert anzugeben (z.B. Records oder Arrays). Sogar offene Typen wie z.B. Strings sind erlaubt. Einer solchen Funktion lassen sich sogar direkt Parameter an REF-Parameter übergeben:

```
Str3:=Concat("foo ",Seg(str1,4,5),
             Seg(str2,6,7)," bar");
```

Prozedurparameter kann man vordefinieren. Beim eigentlichen Aufruf müssen sie nicht angegeben werden. Falls doch, geschieht das gezielt über den Namen:

```
PROCEDURE Test(x,y : INTEGER := 0;
               flag : BOOLEAN :=TRUE);
Test;Test(1);
Test(1,2);
Test(flag:=FALSE);...
```

Außerdem lassen sich Prozeduren mit beliebig vielen Parametern deklarieren:

```
PROCEDURE
    WriteStrings(REF strs : LIST OF STRING);
```

Intern wird ein Parameter vom Typ LIST OF als ein offenes Array repräsentiert:

```
PROCEDURE
    WriteStrings(REF strs : LIST OF STRING);
VAR i : INTEGER;
BEGIN
  FOR i:=0 TO strs'MAX DO
    WriteString(strs[i]);
  END
END WriteStrings;
```

So ist die Prozedur aufzurufen:

```
WriteStrings("Dies ist ein Test");
```

oder

```
WriteStrings("Test Nummer",IntToString(i));
```

Attribut	Bedeutung
'SIZE	Göße in Bytes
'PTR	Liefert Pointer auf die Variable
'ADR	Liefert Adressnummer als LONGINT
'RANGE	Liefert bei offenen Arrays und Strings die Maximallänge
'MIN	Liefert Minimalwert eines Typs.
'MAX	Liefert Maximalwert oder Nummer des höchsten Eintrags eines Arrays oder Strings

Bild 4: Eine der Eigenschaften von Cluster sind die Attribute für Typen und Variablen

Cluster bietet auch einige maschinennahe Elemente. Sie sollte man aber immer in »Low-Level«-Module kapseln. Zum einen existiert ein Inline-Assembler, über den man auf außerhalb gelegene Variablen zugreifen kann (MOVE rec.i,D0), der darüber hinaus auf Wunsch einen Typcheck durchführt. Auch die Condition-Codes des Prozessors lassen sich direkt abfragen:

```
REPEAT
  DEC(i)
UNTIL = ;
```

Die Schleife terminiert erst dann, wenn das Zero-Flag gesetzt ist. Dieser oder ähnlicher Code sollte nur in Low-Level-Modulen Verwendung finden.

Falls eine Pointer-Variable in einem Register liegt, ist es möglich, ohne Umwege den Postincrement- bzw. Predecrement-Modus des MC68000-Prozessors auszunutzen:

```
p1+^:=p2+^.
```

Diese Anweisung schreibt den Wert, auf den p2 zeigt, in die Adresse von p1. Beide Pointer werden anschließend um die Länge ihres Typs erhöht.

Ausnahmebehandlung durch »TRY...EXCEPT« und »RAISE/ASSERT«

Cluster ist die bisher einzige Sprache auf dem Amiga, die dieses mächtige Konzept bietet. Es ermöglicht erstmals auf die Rückgabewerte zu verzichten, die lediglich anzeigen, ob eine Prozedur erfolgreich war oder nicht. Somit entfallen eine Menge lästiger Vergleiche. Erreicht wird das durch selbstdefinierte Exceptions (Ausnahmebehandlungen), die Prozeduren im Fehlerfall setzen. Die Exception ist außerhalb abzufangen und dementsprechend zu reagieren. Dies geschieht zum einen durch »RAISE« und

»ASSERT«, zum anderen durch »TRY..EXCEPT«. Beim Einlesen einer Datei ist es z.B. nicht notwendig, nach jedem Lesevorgang einen Vergleich auf Erfolg durchzuführen, da beim Dateiende die Exception »EOF« ausgelöst wird und das Programm sofort in den »EXCEPT«-Teil verzweigt. Mit Hilfe der TRY..EXCEPT-Anweisung lassen sich auch alle System-Laufzeitfehler behandeln, z.B. die unbeliebte Division durch Null. Eine Exception wird so definiert:

```
EXCEPTION
  Div0    = "Division durch 0";
```

Den angegebenen String erhält man als Laufzeitmeldung, sofern das Abfangen der Exception nicht möglich war.

Ausgelöst wird sie auf zwei Arten:

❒ RAISE(Exceptionname)

❒ ASSERT((Boolesche Bedingung), Exceptionname)

ASSERT löst die Exception nur dann aus, wenn die Boolesche Bedingung unwahr ist.

```
ASSERT(memSize>1000,LowMemory);
```

Ressource-Verwaltung

Cluster benötigt keinen sog. Garbage-Collector, über den sich Ressourcen komfortabel und sicher verwalten lassen. Cluster bietet ein System, mit dem sich alle Ressourcen (Speicher, Dateien usw.) zu sog. Kontexten zusammenfassen lassen, die nach Gebrauch in einem Rutsch freigegeben werden. Es ist möglich, bei jeglicher Anforderung einen eigenen Kontext anzugeben. Tut man das nicht, hängt Cluster diesen dem Aktuellen an und gibt ihn am Programmende automatisch frei. Weiß man, über welche Ressourcen man verfügt und gibt diese dem System zurück, bietet sich das »TRACK..CLOSE..END«-Konstrukt an:

```
TRACK
  Allozierungen
CLOSE
  Anweisungen
END
```

Sie erzeugt einen neuen Kontext, erklärt ihn zum aktuellen und gibt ihn wieder frei. Anschließend ist der vorherige Kontext wieder der aktuelle. Der Clou: Der Kontext wird auch dann zurückgesetzt, wenn eine Exception oder ein Laufzeitfehler auftritt. Der CLOSE-Teil wird immer durchlaufen, unabhängig davon, ob eine Exception vorkam oder nicht.

Eine beliebte Variante ist ein »TRACK..END«-Konstrukt, umgeben von »TRY..EXPECT«. Tritt eine Exception auf, muß man sich ums Freigeben nicht kümmern. Alternativ lassen sich Kontexte auch strukturbezogen verwenden. Wenn man z.B. alle Elemente einer Struktur kontextrelativ angibt, lassen sich diese durch einfaches Freigeben des Kontexts ebenfalls dem System zur Verfügung stellen. Das Ganze funktioniert im Unterschied zu einem Garbage-Collector ohne Performance-Einbußen. Ein weiterer Vorteil ist, daß sich so nicht nur Speicher, sondern auch andere System-Ressourcen durchs Laufzeitsystem verwalten lassen.

Generische Module

Hierbei handelt es sich um Fähigkeiten, Module unabhängig von Typen zu beschreiben. Beispiel: Ein typsicheres Modul zur Verwaltung verketteter Listen ließ sich bislang nur für einen Typ implementieren. Die Verwendung mit einem anderen ähnlichen Typ stieß auf Ablehnung beim Compiler. Es blieb nicht anderes übrig, als den Quelltext zu modifizieren und neu zu kompilieren. Es ergab sich, daß ein Modul in mehreren Varianten existierte. Generische Module umgehen das Problem, indem der Compiler den Code nur einmal einbindet und dennoch durch Ausprägung der volle Typcheck gegeben ist.

Cluster ist darüber hinaus voll objektorientiert, d.h. es verfügt über Klassen, Objekte, Methoden, Vererbung, Generizität mit Objekten, dynamisches Binden, Mehrfacherben etc. Die Vorteile und Eigenschaften des objektorientierten Ansatzes zu erklären, sprengt den Rahmen unseres Artikels. Auf der Cluster-Demo, die als Diskette zum Heft existiert (Seite 114), finden Sie einen ausführlichen Aufsatz über objektorientiertes Programmieren mit Cluster.

Wer gleichzeitig mehrere Versionen eines Programms benötigt, dem bietet Cluster die Möglichkeit, bedingt zu kompilieren. Mit Hilfe von Schaltern (Flags) läßt sich festlegen, welche Programmteile zu übersetzen sind.

Systemeinbindung

Zu allen Amiga-Libraries existieren Cluster-Schnittstellenmodule. Sie besitzen den Vorteil, alle Libraries automatisch zu öffnen und am Programmende wieder zu schließen. In C muß man das oft zu Fuß erledigen.

Sogar TAG-Listen unterstützt Cluster mit dem TAG-Typkonstruktor. Dabei werden sowohl die Tags als auch Tag-Werte überprüft, ob sie zur Funktion passen.

Nach den theoretischen Ausführungen finden Sie abschließend ein Listing, das den Inhalt eines Verzeichnisses ausgibt und die Arbeitsweise von Cluster verdeutlicht. *rz*

Quell- und Literaturhinweise

[1] Von ARexx bis Pascal – Binäre Inflation, AMIGA-Magazin 11/92, Seite 172 ff.

```
 1: | Dieses Programm gibt den Inhalt eines Verzeichnisses als
 2: | Baum aus. Es Erwartet ein Argument über die Shell
 3: MODULE DirTree;
 4:
 5: FROM Arguments  IMPORT Arg,NumArgs;
 6: FROM InOut      IMPORT WriteString,WriteLn,WriteInt;
 7: FROM DosSupport IMPORT DirTreePtr,byName;
 8: FROM Strings    IMPORT Dup;
 9:
10: VAR tree : DirTreePtr; | Zeiger einen Directorybaum,
11:                        | dieser Typ ist im Modul
12:                        | DosSupport definiert.
13:
14: | Ausgabe des Directorybaums
15: PROCEDURE ShowTree(t : DirTreePtr);
16: VAR depth : INTEGER := 0;
17:   | Gibt den Eintragsnamen und erhöht die einrücktiefe,
18:   | für den Fall, daß er ein Unterverzeichnis ist,
19:   | ist dies nicht der Fall, wird dies von After wieder
20:   | rückgängig gemacht.
21:   PROCEDURE Before(t : DirTreePtr);
22:   BEGIN
23:     WriteString(Dup("| ",depth));| Gibt depth-Mal "|" aus.
24:     WriteString(t.data.name^);   | Gibt den Namen aus,
25:     | Füllt die Spalte mit Leezeichen auf.
26:     WriteString(Dup(" ",50-2*depth-t.data.name^.len));
27:     IF t.data.dir THEN
28:       | Falls es ein Verzeichnis ist, wird dessen Größe
29:       | ausgegeben
30:       WriteInt(t.dirData.fullSize,8);
31:     ELSE
32:       | Falls es ein File ist, dessen Länge.
33:       WriteInt(t.data.length,8);
34:     END;
35:     WriteLn;
36:     INC(depth);
37:   END Before;
38:
39:   | Veringert die Einrücktiefe, da man ja wieder ein
40:   | Unterverzeichnis höher ist.
41:   PROCEDURE After(t : DirTreePtr);
42:   BEGIN
43:     DEC(depth);
44:   END After;
45:
46: BEGIN
47:   | Wendet auf alle Elemente des Baumes die Prozedur
48:   | Before an, bevor mit den Söhnen des Baumes
49:   | weitergemacht wird. After wird aufgerufen, nachdem
50:   | der Sohn bearbeitet worden ist. In unserem Fall ist
51:   | jeder Sohn ein Unterverzeichnis
52:
53:   t.ApplyDepthFirstBig(Before,NIL,After);
54: END ShowTree;
55:
56: BEGIN
57:   IF NumArgs=1 THEN     | Check, ob Argument übergeben wurde
58:     tree.Get(Arg(0));   | Directory in Baum einlesen, wobei das
59:                         | 0-te Argument als Pfad verwendet wird.
60:     tree.Sort(byName);  | Sortieren des Baumes nach Namen.
61:     ShowTree(tree);     | Ausgabe des Baumes,
62:     tree.Delete;        | Löschen des Baumes
63:   ELSE
64:     WriteString("DirTree <Pfad>"); WriteLn;
65:   END;
66: END DirTree.     © 1993 M&T
```

Dirtree.mod: Ein einfaches Programm, das den Inhalt eines Verzeichnisses ausgibt. Der Verzeichnisname ist als Argument zu übergeben.

Richtlinien

Systemkonforme Programmierung

Ein Multitasking-Betriebssystem verlangt vom Programmierer die Einhaltung bestimmter Richtlinien – ansonsten verpuffen die Vorteile sang- und klanglos im Nebel der Regelverstöße. Hier die Knackpunkte, worauf bei der Programmierung zu achten ist.

von Rainer Zeitler

Jeder Amiga-Programmierer ist gut beraten, die von Commodore dogmatisierten Richtlinien strikt einzuhalten. Ansonsten riskiert man Inkompatibilitäten mit künftigen Betriebssystemversionen. Die jüngste Vergangenheit hat die Misere deutlich gemacht – Anwender mußten damit rechnen, daß seine Programme unter anderen Versionen nicht laufen.

Programmierrichtlinien sind eine Seite, die andere Kompatibilitätsprobleme, die mit dem Erscheinen neuer Betriebssysteme auftreten. Der gravierendste Einschnitt geschah mit dem Sprung von Betriebssystem 1.3 auf 2.0. Spätestens hier trennte sich die Spreu vom Weizen, wenn das eine oder andere Programm partout nicht funktionieren wollte. Anders beim Betriebssystem 3.0, das im Amiga 1200/4000 eingebaut ist. Hier darf man davon ausgehen, daß nahezu jede Software, die unter 2.0 einwandfrei funktioniert, auch unter 3.0 läuft. Eine pauschale Aussage läßt sich dennoch nicht treffen.

Generelle Programmier-Richtlinien

❒ Schon Einsteiger kennen einen der schlimmsten Verstöße: Die Verwendung von festen Betriebssystemadressen. Wer sich in einem Multitasking-Betriebssystem darauf verläßt, daß bestimmte Funktionen immer an der gleichen Stelle im ROM liegen, ist verlassen. So abgedroschen diese Binsenweisheit auch klingen mag – hier liegt oft die Ursache für fehlerhafte Programme. Die einzige feste Betriebssystemadresse ist 4, über die der Einsprung in die »exec.library« geschieht. Von dort lassen sich alle Libraries, Ressourcen etc. aufspüren.

❒ Ein weiterer, oft anzutreffender Fehler ist die Verwendung von nichtinitialisierten Zeigern, die meist auf die Speicherstelle Null zeigen. Deshalb gilt: Überprüfen Sie vor jedem Schreibzugriff via Zeiger, ob er gefahrlos erfolgen kann.

❒ Es ist zwingend erforderlich, Speicher oder Ressourcen nach Anforderung auch daraufhin zu überprüfen, ob der Aufruf erfolgreich verlief. Man sieht es nur allzu häufig, daß ohne Verifikation vorausgesetzt wird, daß der Speicher oder ähnliches bestimmt zur Verfügung steht. Vorsicht!

❒ Geben Sie alle Ressourcen (Speicher, Libraries etc.) am Programmende wieder frei. Unter Betriebssystem 2.0 und höher ist das im übrigen leicht feststellbar: Gehen Sie ins Shell/CLI und rufen Sie den Befehl AVAIL mit dem Argument »flush« auf. AVAIL zeigt den verfügbaren Speicher an, Flush bewirkt zudem, daß alle überflüssigen Libraries aus dem Speicher entfernt werden. Starten Sie jetzt Ihr Programm. Nach Beendigung tippen Sie erneut »avail flush« ein. Die angezeigten Werte müssen, sofern alle Ressourcen ordentlich dem System zurückgegeben wurden, mit den zuvor ermittelten übereinstimmen. Doch auch wer kein OS 2.0 besitzt, hat eine ähnliche Möglichkeit. Starten Sie hierzu die Workbench mit dem Argument »-debug«. Sie werden einen neuen Menüpunkt »Flushlibs« entdecken, der ebendies tut – er entfernt nicht benutzte Libraries aus dem Speicher. Ein AVAIL-Aufruf bringt dann Klarheit.

❒ Erstaunlicherweise tauchen häufig Systemabstürze bei 32-Bit-Prozessoren auf, die z.B. im Amiga 1200/3000/4000 zu finden sind. Gerade bei Assembler-Programmen ist das zu beobachten. Sie laufen prächtig unter Prozessoren, die lediglich Speicherbereiche mit 24 Bit adressieren (z.B. Amiga 500/600/2000), da hier die oberen acht Bit unbeachtet bleiben. Nicht so beim 32-Bit-Prozessor. Nur ein falsches Bit, und die Katastrophe ist unvermeidlich. Deshalb: Struktur- und Variablenbereiche sollten vor der Benutzung immer mit Null belegt werden. Zugegeben, Hochsprachenprogrammierer haben's hier einfacher, da diese Arbeit schon von den meisten Compilern abgenommen wird. In diesem Zusammenhang sei auf folgende Fehlerquellen hingewiesen: Vergleichen Sie auf 32-Bit-Maschinen Adressen niemals mit vorzeichenbehafteten Zahlen. Auch der Test auf einen Null-Zeiger mit 8- oder 16-Bit-Variablen ist tabu. Verwenden Sie immer Variablen vom Typ »unsigned long«.

❒ Und wieder trifft's die Assembler-Programmierer. Einem Drahtseilakt vergleichbar ist die Angewohnheit, nach einem Funktionsaufruf des Betriebssystems davon auszugehen, daß die Register D0, D1, A0 und A1 unverändert sind. Auch wenn es bei einigen so ist – in Zukunft muß das nicht so sein. Aufgrund überraschender Fehlfunktionen vieler Programme modifizierten die Commodore-Entwickler schon abgeschlossene Systemfunktionen insofern, daß die oben genannten Register nach dem Aufruf wieder den gleichen Inhalt besaßen. Verlassen Sie sich in künftigen Betriebssystemen nicht darauf, daß ein Betriebssystem an Programme angepaßt wird und nicht umgekehrt – paradox ist das Ganze jedenfalls schon.

❒ Wenn wir schon bei Assembler-Programmieren sind: Rufen Sie niemals Betriebssystemfunktionen auf, ohne zuvor den Library-Vektor ins sechste Adreßregister (A6) befördert zu haben. Schließlich gehen alle Funktionen davon aus, hier die Library-Adresse zu finden. Das gilt auch für Funktionen, die den Library-Pointer eigentlich nicht benötigen, denn auch das kann in spe ganz anders aussehen.

❒ Einige Funktionen älterer Betriebssytemversionen wiesen zwar den einen oder anderen Parameter auf – genutzt wurde er dort jedoch nicht. Das ändert sich aber so sicher wie das Amen in der Kirche. Wenn auch nicht jetzt, so doch bei späteren Versionen. Es ist außerordentlich wichtig, diese Parameter korrekt zu initialisieren – und wenn's »nur« eine Null ist. Ansonsten kann es später zu unangenehmen Folgen führen.

Für Programmierer sind die Richtlinien bindend

❒ Verwenden Sie niemals mit »private« gekennzeichnete Elemente von Systemstrukturen (z.B. die »intuition.Library« von OS 1.3). Das kann einmal gutgehen, ein anderes Mal nicht, da das Element z.B. für gänzlich andere Aufgaben als ursprünglich geplant eingesetzt wird.

❒ Besondere Vorsicht ist bei Programmen mit selbstmodifizierendem Code geboten. Der MC68040-Prozessor verfügt über einen 4 KByte großen Daten- und Instruction-Cache. Im schlimmsten Fall greift er auf nicht existente Daten zu ([1]). Spezielle Funktionen der »exec.library«, z.B. CacheClearU() oder CachePostDMA(), beugen diesen Fehlern vor. Gleiches gilt fürs Trackdisk-Device – vor dem Lesen ist der Puffer zu leeren (Flush).

❒ Wenn Sie Zeitintervalle oder Pausen in Ihrem Programm benötigen – verwirklichen Sie das niemals mit Hilfe von Schleifen, die soundso oft ausgeführt werden. Erstens steht das mit dem Multitasking nicht in Einklang, da die Rechenleistung des Amiga in die Knie geht, zum anderen sind die Pausen auf dem Amiga 3000 kürzer als die auf einem Amiga 500. Klar, ist der Prozessor doch um einiges fixer. Greifen Sie stattdessen auf die elegan-

ten Möglichkeiten des Timer-Device zurück. Es bietet die erforderlichen Funktionen und wird auch in Zukunft funktionieren.

❐ Greifen Sie auf Hardwareregister zu, initialisieren Sie diese mit Ihren eigenen Einstellungen. Gehen Sie niemals von voreingestellten Werten aus, da sich diese unter neuen Betriebssystemversionen ändern können.

❐ ASL-Library hin, ARP-Library her: Seit OS 2.0 gibt's dank der ASL-Library einheitliche Datei- und Font-Requester. Demnach ist es überflüssig, eigene zu programmieren, denn sie sind extrem leistungsfähig. Sicher, es gibt zig weitere PD-Libraries, die hier und da ihre Vorzüge besitzen; dennoch sollten die Programme auf die ASL-Library umgestellt werden oder wenigstens eine Option bieten, den benutzten Requester zu wählen.

Kompatibilitätsrisiken unter OS 2.0

Bei der Umstellung von OS 1.3 auf 2.0 gab's viele Veränderungen. Einige der wichtigsten stellen wir vor:

❐ Amiga-DOS wurde komplett überarbeitet und hat die BCPL-Relikte hinter sich gelassen. BCPL brachte die Eigenschaft mit sich, daß DOS-Funktionen Rückgaben sowohl übers Register D0 als auch D1 lieferten. Das ist passé. Bis auf wenige Ausnahmen steht der Return-Wert nur noch in D0.

❐ Das Audio-Device wird erst dann in den Speicher geladen, wenn es benötigt wird. Unter Umständen kann das aufgrund mangelnden Speichers schiefgehen. Nach dem ersten Öffnen und trotz korrekter Freigabe aller Ressourcen verlieren Sie Speicher. Beim zweiten Aufruf passiert das nicht.

❐ Möchten Sie per Software einen Reset auslösen, war es unter OS 1.3 (und älter) üblich, an den Anfang des ROM-Bereichs (0xFC0002) zu springen. Unter dem Amiga 3000 und unter späteren Betriebssystemversionen funktioniert das nicht mehr. Benutzen Sie stattdessen die im Listing abgedruckte Assembler-Routine. Sie wird von Commodore offiziell vorgeschlagen.

❐ Die »intuition.library« hat sich grundlegend geändert. Wenn Sie Fenster auf der Workbench öffnen, bedenken Sie, daß der Workbench-Screen andere Dimensionen, mehr Farben und andere Zeichensätze haben kann. Außerdem ist es jetzt möglich, daß ein Screen negative Ursprungskoordinaten besitzt. Den verwendeten Zeichensatz identifizieren Sie über den Screenpointer der Workbench:

```
struct Screen *WBScreen;
// Der verwendete Zeichensatz
struct Font   *Screenfont=
        WBScreen->RastPort.Font;
// Die Höhe des Zeichensatzes
WORD    FontHoehe =
        WBScreen->RastPort.TxHeight;
```

Die Textbreite kann variieren, falls proportionaler Zeichensatz eingestellt ist. Doch es steht immer auch nichtproportionaler Zeichensatz zur Verfügung, der via Preferences einzustellen ist. Über die »graphics.library« ist dieser zu erfahren:

```
; Die von Commodore empfohlene Resetroutine

    INCLUDE "exec/types.i"
    INCLUDE "exec/libraries.i"

    csect  text
    xdef   _ColdReboot
    xref   _LVOSupervisor

REALEXECBASE     equ 4          ; Pointer exec.library
MAGIC_ROMEND     equ $01000000  ; Ende des Kickstart-ROM
MAGIC_OFFSET     equ -$14       ; Offset vom ROM-Ende zur
                                ; Kickstart-Größe
KICK_V36         equ 36         ; Ab Kickstart V36 stellt das
                                ; Betriebssystem eine eigene
                                ; Reset-Routine zur Verfügung
V36_ColdReboot   equ -726       ; Offset der Reset-Routine

_ColdReboot:
    move.l  REALEXECBASE,a6     ; exec.library-Pointer in A6
    cmp.w   #KICK_V36,LIB_VERSION(a6)
    blt.s   old_exec            ; Kickstartversion < 36
                                ; Der Reset muß selbst aus-
                                ; geführt werden
    jmp     V36_ColdReboot(a6)  ; Sonst die Coldreboot-
                                ; Routine anspringen, aus der
                                ; wir niemals zurückkehren

old_exec:
    lea.l   GoAway(pc),a5       ; Die Adresse des Reset-Codes
    jsr     _LVOSupervisor(a6)  ; Den Code ausführen
    ; Diesen Punkt erreichen wir nicht mehr
    CNOP    0,4                 ; Auf Langwort ausrichten

GoAway:
    lea.l   MAGIC_ROMEND,a0     ; ROM-Ende
    sub.l   MAGIC_OFFSET(a0),a0; berechnen des PC
    move.l  4(a0),a0            ; PC beim Einsprung
    subq.l  #2,a0               ; Zeigt auf den zweiten Reset
    reset                       ; reset und jmp müssen ein
    jmp     (a0)                ; Langwort teilen
    END
```

So geht's: Der richtige Reset per Software ist problemlos möglich – am besten mit der von Commodore empfohlenen Routine.

```
// Zeiger auf graphics.library
struct GfxBase *GfxBase;
// Der eingestellte Monospaced-Font
struct Font
*DefaultFont=GfxBase->DefaultFont;
// Die Höhe des Zeichensatzes
WORD Fonthoehe=
GfxBase->DefaultFont->tf_YSize;
// Die Breite eines Zeichens
WORD Fontbreite=
GfxBase->DefaultFont->tf_XSize;
```

Gehen Sie niemals davon aus, daß der Zeichensatz Topaz 8 eingestellt ist. Tun Sie das dennoch, ergeben sich unschöne Effekte und der Anwender muß, nur um Ihr Programm bedienen zu können, Topaz 8 extra wählen.

❐ Verwenden Sie zudem niemals in Ihrer Applikation eine Tastenkombination mit der linken Amiga-Taste. Diese ist für Systemzwecke reserviert.

❐ Auch die oft benutzte Funktion GetScreenData(), mit deren Hilfe sich die Workbench-Auflösung und -Dimension einfach kopieren ließ, ist nicht empfehlenswert. Benutzen Sie stattdessen die Funktionen LockPubScreen() und GetVPModeID(), um die Workbench-Daten in Erfahrung zu bringen.

❐ Die gemeinsame Angabe der IDCMP-Messageflags RAWKEY und VANILLAKEY erzeugt jetzt eine Message vom Typ RAWKEY, wenn eine Sondertaste betätigt wurde, sonst VANILLAKEY. Der Umweg über die Funktion RawKeyKonvert() entfällt.

❐ Auch bei der »graphics.library« sind einige Dinge zu beachten. Die Einträge »DisplayFlags«, »row« und »cols« müssen nicht mit denen der Workbench übereinstimmen. Die ColorMap-Strukur ist größer und sollte mit der Funktion GetColorMap() generiert werden. Eine beliebte Methode, Bildschirmpixel anzusprechen, war die Anwendung der Modulo-Funktion auf die Bitmap-Breite. Das kann schiefgehen. Sicherer ist es, den Wert »BitMap->BytesPerRow« zu nehmen. Im übrigen ist es für zukünftige Betriebssysteme außerordentlich wichtig, die Bitmap nicht über den Eintrag in der Screen-Struktur zu erreichen, sondern über den Rastport:

```
// Falsch
struct BitMap *bmap=Screen->BitMap;
// Richtig
struct BitMap
*bmap=Screen->RastPort.BitMap;
```

Zu Inkompatibilitäten kann auch die Allokierung von Bitmaps mit der AllocMem()-Funktion führen. Das Reservieren benötigten Speichers für Bitmaps ist grundsätzlich mit der AllocRaster()-Funktion vorzunehmen.

Sicher, es gibt weitere mögliche Fehlerquellen. Beachtet man jedoch die hier vorgestellten, spart man sich bereits eine Menge – unnötigen – Ärger. ■

Literaturhinweise:

[1] Zeitler, Rainer: Programmieren unter OS 2.0 – die »exec.library«, AMIGA-Magazin 6/92, Seite 124

[2] Commodore-Amiga, ROM Kernel Reference Manual »Hardware«, Third Edition, Addison Wesley

Huffman-Codes

Datenkomprimierung in Assembler

von Sebastian Wedeniwski

Das Datenkompressionsverfahren von D.Huffman kann in Textdateien (und vielen anderen Arten von Dateien) eine beträchtliche Platzeinsparung ermöglichen. Die Huffman-Kodierung wurde 1952 entdeckt, wobei die Implementation modernere algorithmische Techniken verwendet. Folgende Idee steckt hinter dem Verfahren:

Anstatt die üblichen acht Bits für jedes Zeichen zu benutzen, werden für Zeichen, die häufig auftreten, nur wenige Bits verwendet, und mehr Bits für die, die selten vorkommen.

Bei der Kodierung darf kein Zeichencode mit dem Anfang eines anderen übereinstimmen, um nur eine Möglichkeit bei der Dekodierung zu gewährleisten (z.B. »A« mit 1, »B« mit 01 und »C« mit 00 verschlüsseln). Eine einfache Methode zur Darstellung der Codes ist die Verwendung eines Trie.

Gut gepackt, ist viel gewonnen

Ein Trie ist ein modifiziertes Modell des digitalen Baums, in dem die Schlüssel nicht in Knoten des Baums gespeichert, sondern statt dessen in äußeren Knoten des Baums angeordnet werden. Infolgedessen sind zwei Typen von Knoten vorhanden:
- Innere Knoten, die nur Verkettungen zu anderen Knoten implizieren, und
- äußere Knoten, die Schlüssel und keine Verkettungen implizieren.

In dieser Struktur wird der Schlüssel zufolge seinen Bits abgezweigt (bei 0 nach »links« und bei 1 nach »rechts«). Ein digitaler Trie für das obenerwähnte Beispiel:

ABCBA
kodiert zu: 10100011

Die einzelnen Schritte bei der Erzeugung des Huffman-Codes:

Im ersten Schritt wird durch Zählen der Häufigkeit jedes Zeichens innerhalb der zu kodierenden Zeichenfolge ermittelt. Der nächste Schritt ist dann der Aufbau des Kodierungs-Tries gemäß der Häufigkeit. Der Trie wird durch einen binären Baum mit Häufigkeit, die in den Knoten gespeichert sind, generiert: Vorerst wird für jede von null verschiedene Häufigkeit ein Knoten des Baums generiert. Anschließend werden die beiden Knoten mit den kleinsten Häufigkeiten ausgesucht und es wird ein neuer Knoten generiert, der diese beiden Knoten als Nachfolger hat und dessen Häufigkeit einen Wert hat, der gleich der Summe der Werte für seine Nachfolger ist. Danach werden die beiden Knoten mit der kleinsten Häufigkeit in diesem Wald ermittelt und ein neuer Knoten wird auf diese Weise generiert. Letztendlich sind alle Knoten miteinander zu einem einzigen Baum vereinigt.

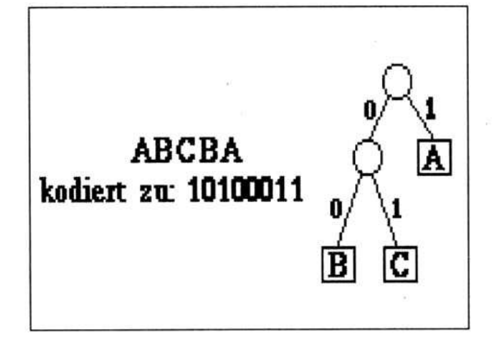


Bild: Ein Trie ist eine Abwandlung eines binären Baums

Jetzt kann der Huffman-Code abgeleitet werden, indem die Häufigkeiten an den unteren Knoten einfach durch die zugehörigen Buchstaben substituiert werden und der Baum dann als ein Trie für die Kodierung betrachtet wird. Die vorgestellte Implementation ist so abgestimmt, daß sie ohne weiteres in eigene Programme eingebaut werden kann. Diese Implementation besteht aus zwei Teilen, der eine Teil »Pack« komprimiert den vorgegebenen Speicherbereich und der zweite Teil »Unpack« entschlüsselt den Huffman-Code und schreibt des Resultat in einen vorgegebenen Speicherbereich. Die Handhabung läuft wie folgt ab:

Komprimierung: Die Anfangsadresse des Speicherbereichs, der komprimiert werden soll, muß im Adreßregister a1 und die Anfangsadresse des Resultats in a2 stehen. Die Länge des zu komprimierenden Speicherbereichs muß im Datenregister d7 stehen (Tip: d7 sollte zum komprimierten Programm beigefügt werden, um den Speicherumfang beim Entpacken zu kennen). Nach dem Aufruf von »Pack« müssen zum Resultat die Kodierungstabellen »length« (Länge des komprimierten Speicherbereichs in Bits), »code« (Huffman-Code für jedes Byte) und »len« (Länge des entsprechenden »code«) beigefügt werden. Diese Codes sind elementar beim Entpacken.

Entpacken: Die Anfangsadresse der komprimierten Datei muß im Adreßregister a0 und die Anfangsadresse des Resultats muß in a5 stehen. Jetzt kann »Unpack« aufgerufen werden.

Das Datenkomprimierprogramm ist schnell (je nach Speicherumfang) und effizient. Zum Vergleich haben wir ein Testbild mit der Länge von 49428 Byte komprimiert und kamen auf 18916 + 772 (Kodierungstabelle) Byte. Der »IFF-Packer« kam bei der gleichen Datei auf 21786 Byte, d.h. benötigte rund 20 Prozent mehr Platz für die gepackte Datei. ub

```
 1:  lea a,a1  ; Anfangsadresse der Quelldatei
 2:  lea ziel,a2  ; Anfangsadresse der Zieldatei
 3:  move.l #aend-a,d7 ; Quelldateilänge
 4:  jsr Pack
 5:  lea ziel,a0  ; Anfangsadresse der komprimie
    rten Datei
 6:  lea a,a5  ; Anfangsadresse der Zieldatei
 7:  jsr Unpack
 8:  rts
 9: a: incbin "Programmname" ; zu komprimierende
    s Programm
10: aend:
11: Pack:
12:  move.l d7,-(sp)
13:  move.l a1,-(sp)
14:  move.l a2,-(sp)
15:  lea count(pc),a0
16:  subq.l #1,d7
17: Anzahl: moveq #0,d0
18:  move.b (a1,d7.l),d0
19:  add.w d0,d0
20:  add.w d0,d0
21:  addq.l #1,(a0,d0.w)
22:  dbra d7,Anzahl
23:  moveq #0,d0
24:  move.l d0,d7
25:  move.l a0,a2
26:  lea heap(pc),a3
27: Loop2: tst.l (a2)+
28:  beq.s Loop
29:  addq.w #2,d0
30:  move.w d7,(a3)+
31: Loop: addq.w #4,d7
32:  cmp.w #$400,d7
33:  bne.s Loop2
34:  lea heap(pc),a3
35:  move.l d0,d6
36:  move.l d0,d1
37:  lsr.w #1,d1
38: down: move.w d6,d7
39:  move.w -2(a3,d7.w),d2
40:  bsr downheap
41:  subq.w #2,d6
42:  bne.s down
43:  lea dad(pc),a4
44: repeat: move.w (a3),d6
45:  move.w -2(a3,d0.w),(a3)
46:  subq.w #2,d0
47:  moveq #2,d7
48:  move.l d0,d1
49:  lsr.w #1,d1
50:  move.w (a3),d2
```

```
51:  bsr downheap
52:  move.w (a3),d2
53:  move.l (a0,d2.w),d3
54:  add.l (a0,d6.w),d3
55:  move.l d0,d7
56:  add.l d7,d7
57:  move.l d3,-4(a2,d7.w)
58:  add.l #$3fc,d7
59:  move.l d7,(a4,d6.w)
60:  move.w d7,(a3)
61:  neg.l d7
62:  move.l d7,(a4,d2.w)
63:  moveq #2,d7
64:  move.w (a3),d2
65:  bsr downheap
66:  cmp.w #2,d0
67:  bne.s repeat
68:  clr.l $400(a4)
69:  lea code(pc),a2
70:  lea len(pc),a3
71:  moveq #0,d7
72: for: moveq #-1,d0
73:  tst.l (a0)+
74:  bne.s Weiter2
75:  clr.w (a2)+
76:  move.b d0,(a3)+
77:  bra.s next
78: Weiter2:moveq #1,d1
79:  moveq #0,d3
80:  move.l (a4,d7.w),d2
81: repeat2:bge.s ben
82:  add.l d1,d3
83:  neg.l d2
84: ben: add.l d1,d1
85:  addq.l #1,d0
86:  move.l (a4,d2.w),d2
87:  bne.s repeat2
88:  move.w d3,(a2)+
89:  move.b d0,(a3)+
90: next: addq.w #4,d7
91:  cmp.w #$400,d7
92:  bne.s for
93:  move.l (sp)+,a0
94:  move.l (sp)+,a2
95:  add.l (sp)+,a2
96:  lea len(pc),a3
97:  lea code(pc),a4
98:  moveq #0,d7
99:  move.l d7,d0
100:  move.l d7,d6
101:  move.l d7,d2
102: for2: moveq #0,d0
103:  move.b (a1)+,d0
104:  moveq #0,d1
105:  move.b (a3,d0.w),d1
106:  add.w d0,d0
107:  lea 1(a4,d0.w),a5
108: rueck: cmp.b #8,d1
109:  blt.s rueck2
110:  move.b d1,d2
111:  subq.b #8,d2
112:  btst d2,-1(a5)
113:  bra.s wtw
114: rueck2: btst d1,(a5)
115: wtw: beq.s wt
116:  bset d7,(a0)
117: wt: addq.l #1,d6
118:  addq.b #1,d7
119:  and.b #7,d7
120:  bne.s wet
121:  addq.l #1,a0
122: wet: dbf d1,rueck
123:  cmp.l a1,a2
124:  bne.s for2
125:  subq.l #1,d6
126:  lea length(pc),a0
127:  move.l d6,(a0)
128:  rts
129: downheap:
130:  cmp.w d1,d7
131:  bgt.s Weiter
132:  move.l d7,d3
133:  add.w d3,d3
134:  cmp.w d0,d3
135:  bge.s Sprung
136:  lea -2(a3,d3.w),a5
137:  move.w (a5)+,d4
138:  move.w (a5),d5
139:  move.l (a0,d5.w),d5
140:  cmp.l (a0,d4.w),d5
141:  bge.s Sprung
142:  addq.w #2,d3
143: Sprung: move.w -2(a3,d3.w),d4
144:  move.l (a0,d4.w),d5
145:  cmp.l (a0,d2.w),d5
146:  bgt.s Weiter
147:  move.w d4,-2(a3,d7.w)
148:  move.w d3,d7
149:  bra.s downheap
150: Weiter: move.w d2,-2(a3,d7.w)
151:  rts
152: heap: dcb.w 256,0
153: count: dcb.l 2*256,0
154: dad: dcb.l 2*256,0
155: Unpack:
156:  move.l length(pc),d4 ; komprimierte Dateilä
      nge in Bits
157:  lea code(pc),a3
158:  moveq #0,d0
159:  move.l d0,d6
160:  move.l d0,d5
161: re: btst d6,(a0)
162:  beq.s nicht
163:  addq.b #1,d0
164: nicht: addq.b #1,d6
165:  and.b #7,d6
166:  bne.s nicht2
167:  addq.l #1,a0
168: nicht2: moveq #0,d7
169:  lea len(pc),a2
170: l1: cmp.b (a2)+,d5
171:  bne.s l2
172:  move.w d7,d1
173:  add.w d1,d1
174:  cmp.w (a3,d1.w),d0
175:  bne.s l2
176:  moveq #0,d0
177:  move.b #-1,d5
178:  move.b d7,(a5)+
179:  move.b d5,d7
180: l2: addq.b #1,d7
181:  bne.s l1
182:  add.w d0,d0
183:  addq.b #1,d5
184:  dbf d4,re
185:  rts
186: length: dc.l 0  ; komprimiertes Programm
187: code: dcb.w 256,0
188: len: dcb.b 256,0
189: ziel: dcb.b aend-a,0  ;     © 1993 M&T
```

»Huffman-Codes.asm: Ein Programm zum Packen in Assembler

Angewandte Mathematik

Differenzieren

Das Ableiten von Funktionen ist für viele ein rotes Tuch. Doch gerade in der Mathematik findet man viele Beispiele, komplexe Lösungsverfahren mit dem Computer einfach und effizient umzusetzen. Und zum Glück gehört das Ableiten von Funktionen dazu.

von Markus Öllinger

Ein Computer ist in vielerlei Hinsicht für den Menschen eine Unterstützung. Eine seiner herausragenden Leistungen ist es, Berechnungen in einer für menschliche Verhältnisse ungewöhnlichen Geschwindigkeit durchzuführen. Mit ihm läßt sich viel Zeit für die Lösung schwieriger Rechnungen einsparen. Ein weiterer Vorteil ist die Eigenschaft korrekter Algorithmen (Programme), daß sie immer ein korrektes Ergebnis liefern. Rechenfehler können beim funktionstüchtigen Computer ausgeschlossen werden.

Programme, die mathematische Berechnungen übernehmen, haben meist einen Nachteil: Sie sind kompliziert. Selbst Rechenaufgaben, die für den kopfrechnenden Menschen einfach sind, können einen Programmierer hier und da zur Verzweiflung bringen. Trotzdem lohnt sich meist der Aufwand und deshalb wollen wir uns hier mit einer Rechenart beschäftigen, die wohl jeder fortgeschrittene Gymnasiast schon genossen haben dürfte: dem Differenzieren.

Der Amiga kann von sich aus nicht differenzieren. Wir müssen ihm Schritt für Schritt beibringen, was beim Differenzieren einer Funktion alles zu erledigen ist. Dazu aber müssen wir selbst differenzieren können, ansonsten dürfte es schwer fallen, den Computer via Programmiersprache zu instruieren.

Es ist nicht einfach, den Vorgang in ein Programm zu fassen. Wir wissen jedoch, daß das Differenzieren nach einem bestimmten Schema abläuft. Es muß demzufolge ein Algorithmus existieren, der alle differenzierbaren Funktionen ableiten kann. Das macht Zuversicht. Anders sieht das bei der Umkehrung der Ableitung aus, dem Integrieren. Legen wir also zuerst fest, was unser Programm alles leisten soll:

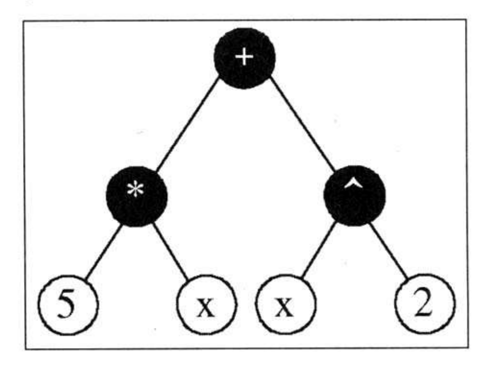
Wir benötigen eine Funktion (nennen wir sie »Diff«), der wir eine Zeichenkette mit der zu differenzierenden Formel übergeben und die uns eine neue Zeichenkette mit der dazugehörigen Ableitung liefert. Unterstützt werden dabei die üblichen mathematischen Operatoren »(«, »)«, »+«, »-«, »*«, »/«, »^« sowie diverse Winkelfunktionen. Es ist klar, daß wir mit der gegebenen Repräsentation der Funktion in Form einer Zeichenkette (String) wenig Freude haben werden. Der erste Schritt besteht nun darin, die Funktion in anderer Form zu speichern, die für unsere Aufgabe besser geeignet ist: als Baum.

Betrachten Sie zunächst das Bild des Binärbaums. Er besteht aus einzelnen Knoten, in der Abbildung als kleine Kreise dargestellt, die durch sog. Kanten miteinander verbunden sind. In der Informatik wachsen Bäume von oben nach unten. Der obere Knoten wird deshalb als Wurzel bezeichnet, die Knoten am unteren Ende als Blätter. Wie Sie sehen, speichern wir die Operanden immer als Blätter und die dazugehörigen Operatoren als innere Knoten. Um nun eine Funktion in einem Baum abzulesen, müssen wir die Knoten von links nach rechts auslesen und haben die ursprüngliche Funktion wieder.

Wir wissen jetzt, wie wir unsere Formel ablegen. Der nächste Schritt ist, einen Algorithmus herzuleiten, der die Konvertierung eines Strings in so einen Binärbaum übernimmt. Selbstverständlich muß dieser Programmteil auch die Prioritäten der mathematischen Operatoren sowie die Auswertung von Klammern berücksichtigen. Wir gehen dazu wie folgt vor: Wir verwenden zwei Stapel, einen für die Operatoren (+, - usw.), einen zweiten für die Operanden, auf die sich die Operatoren beziehen. Wir scannen die Zeichenkette zeichenweise durch. Dabei unterscheiden wir folgende Fälle:

- Wir treffen auf einen Operanden: Wir legen ihn auf den Operanden-Stack.
- Wir treffen auf eine öffnende Klammer »(«: Diese legen wir auf den Operator-Stack.
- Wir treffen auf einen anderen Operator: Hier werden alle Operatoren mit höherer oder gleicher Priorität als der zuvor gefundenen auf dem Operator-Stapel vom Stapel geholt, bis der Stapel leer ist oder eine öffnende Klammer erscheint (»(«). Anschließend wird der neu im String gefundene Operator auf den Operator-Stapel geschoben.
- Nach einer schließenden Klammer »)« werden alle Operatoren unabhängig von ihrer Priorität vom Stapel geholt, bis wir eine öffnende Klammer finden.
- Am Ende der Zeichenkette werden alle Operatoren vom Stack geholt, bis er leer ist.

Mit allen vom Stapel geholten Operatoren wird die Routine »verwende_Operator« aufgerufen. Bei einem binären Operator holt sich



Die Repräsentation der Funktion »5*x+x^2« als binärer Baum

verwende_Operator zwei Operanden vom entsprechenden Stapel, hängt sie als rechten und linken Sohn des Operator-Knotens an und schiebt anschließend den Operator-Knoten auf den Operanden-Stapel. Bei unären Operatoren (z.B. negatives Vorzeichen) wird nur ein Operand geholt und als rechter Sohn an den Operator angehängt. So wird der Baum stückweise zusammengesetzt.

Um die Wirkungsweise dieses Algorithmus zu veranschaulichen, führen wir uns das Beispiel aus unserem Bild vor Augen und konstruieren den dazugehörigen Baum zu Fuß. Die Tabelle zeigt, wie der fertige Baum entsteht. Beachten Sie, daß die Formeln im

TABELLE

Schritt	Operanden-Stapel	Operator-Stapel	Bemerkung
1. 5	5		Operand auf Stapel
2. *	5	*	Operator auf Stapel
3. x	5, x	*	Operand auf Stapel
4. +	5*x		Verwende »*«
		+	Operator auf Stapel
5. x	5*x, x	+	Operand auf Stapel
6. ^	5*x, x	+, ^	Operator auf Stapel
7. 2	5*x, x, 2	+, ^	Operand auf Stapel
8. Stringende	5*x,x^2	+	Verwende »^«
	5*x+x^2		Verwende »+«

Operanden-Stapel keine Zeichenketten, sondern schon Teilbäume sind, die von verwende_Operator gebildet wurden. Aus der Tabelle ist ersichtlich, daß sich der Baum am Ende auf dem Operanden-Stapel befindet. Eine Eigenschaft solcher Bäume ist, daß sie keine Klammern enthalten. Sie sind, wie bei Postfix-Ausdrücken, implizit in der Baumstruktur enthalten, indem geklammerte Ausdrücke einfach tiefer im Baum erscheinen.

Nachdem die Funktion in eine passende Form gebracht ist, wird der Baum zum Differenzieren, beginnend bei der Wurzel, abgearbeitet. Dabei bestimmt der Operatorknoten, welche Ableitungsregel zum Einsatz kommt.

Bevor wir mit der Realisierung beginnen, setzen wir uns zunächst mit dem Problem auseinander, wie bestimmte Operatoren zu differenzieren sind. Wir führen hierzu folgende Konvention ein:

- u repräsentiert immer den linken Operanden eines Operators,
- v immer den rechten.

Beispiel: »5*x« bedeutet, daß u die Zahl »5« repräsentiert, v das Symbol »x«. Weiterhin vereinbaren wir, daß u differenziert a lauten soll, v hingegen b. Beginnen wir mit der Unterscheidung (abhängig vom Operator):

- Wir treffen auf ein »+« oder »-«: Hier haben wir es nicht schwer. Wir differenzieren einfach den Teil links und rechts vom Operator und sind fertig. Aus u+v wird also a+b.
- Wir finden ein »*« vor: Hier kommt die Produktregel zum Einsatz: aus u*v wird a*v+b*u.
- Wir treffen auf ein »/«: Hier verwenden wir die Quotientenregel: (a*v-b*u)/v^2.
- Wir treffen auf ein »^«: Die anzuwendende Regel ist nicht so einfach und fehlt auch in den meisten Formelsammlungen. Deshalb die kurze Herleitung: Zuerst formen wir »u^v« in »exp(ln(u^v))=exp(v*ln u)«. Nun differenzieren wir diesen Ausdruck nach der Ketten- und Produktregel und erhalten: »exp(v*ln u)*(b*ln u+a/u*v)=u^v*(b*ln u+a/u*v)«.

Somit hätten wir die wichtigsten Formeln zum Differenzieren – geordnet nach den einzelnen Operatoren – und können nun die Arbeitsweise des eigentlichen Differenzierteils angehen. Wie Sie gleich sehen werden, handelt es sich dabei um einen rekursiven Algorithmus. Darunter versteht man eine Programmiermethode, bei der sich eine Funktion selbst aufruft. Betrachten wir, wie mit Rekursion der Ableitungsvorgang schnell und relativ einfach zu lösen ist:

Wir beginnen bei der Wurzel des Baums und rufen die Routine »Differenziere« mit der Wurzel als Argument auf. Handelt es sich um eine Konstante oder Variable, wird sie einfach differenziert (Konstante wird 0, Variable x wird 1). Andernfalls differenziert die Routine zuerst ihren linken Teilbaum (entspricht dem Term links vom Operator, also u), anschließend ihren rechten Teilbaum (also v). So erhalten wir a und b, also die Ableitungen von u und v. Wir müssen jetzt nur noch die Terme entsprechend den oben angegebenen Regeln zusammensetzen.

Anschließend ist noch das Umwandeln des so entstandenen Baums in einen String vorzunehmen. Das ist ebenfalls ein rekursiver Vorgang. Prinzipiell sieht die Vorgehensweise so aus, daß wir wieder bei der Wurzel beginnen und zuerst den linken Teilbaum, dann den Knoten selbst und anschließend den rechten Teilbaum ausgeben. Diese Art der Baumausgabe nennt man auch symmetrische Reihenfolge. Damit ist es jedoch noch nicht getan, denn wir müssen auch die (im Baum nicht mehr enthaltenen) Klammern wiederherstellen. Das ist jedoch einfach, reicht es doch aus, die Prioritäten der Operatoren zu vergleichen. In einem Term ohne Klammern besitzt der Vater immer einen Operator niedrigerer Priorität als seine Söhne (Bild). Hat irgendwo im Baum der Operator des Vaters eine höhere Priorität als der des Sohns, ist eine Klammer zu setzen. Bei gleicher Priorität ist nur nach nicht kommutativen Operatoren eine Klammer nötig, um Terme wie »5-(x+2)« richtig zu verarbeiten.

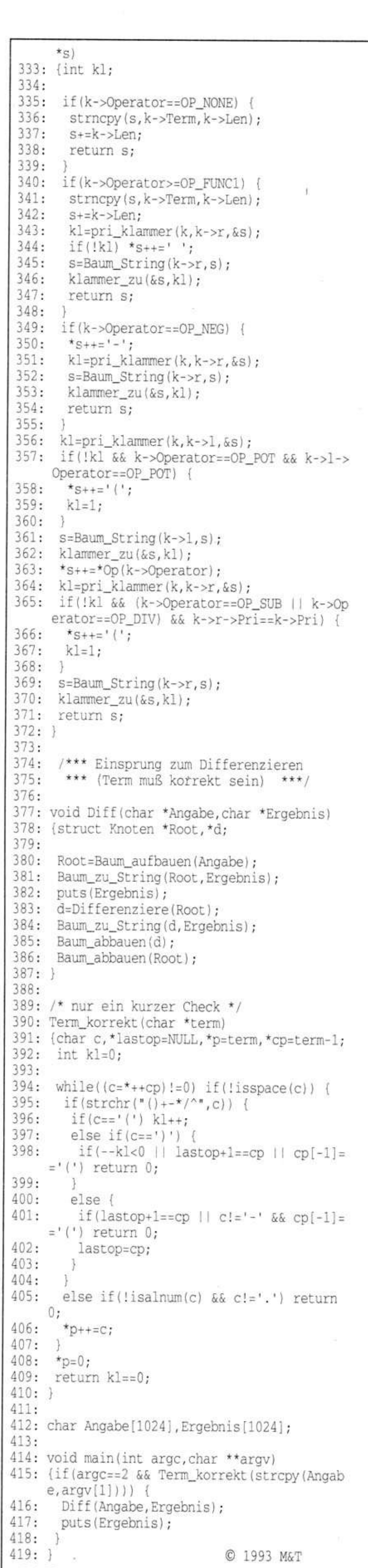
Abschließend werfen wir einen Blick auf unser Beispiellisting, in dem alles Besprochene als Programm verwirklicht ist. Nachdem eine kurze (nicht vollständige) Überprüfung des Terms auf Korrektheit stattgefunden hat, sorgen die Routinen »Baum_aufbauen« und »verwende_Operator« für die Erzeugung des Baums. Anschließend wird die Funktion »Differenziere« aufgerufen, die den Baum ableitet. Interessant ist die Prozedur »Regeln«, die einen Term nach einer von uns hergeleiteten Regeln konstruiert. Die Routine erhält die Differenzierregeln in einer eigenen Schreibweise, der Präfix-Notation. Diese zeichnet sich dadurch aus, daß dabei der Operator immer vor seinen Operanden steht. Der Infix-Term »5*x+x^2« lautet in Präfix-Notation »+*5x^x2«, die Produktregel also »+*av*bu«. Aus so einem Term ist (wiederum per Rekursion) mit Leichtigkeit ein Baum zu erzeugen, da ein Präfixterm genau die Reihenfolge angibt, die der Baum von oben nach unten beschreibt.

Z.Zt. unterstützt das Programm die mathematischen Funktionen ln, exp, sin, cos, sinh und cosh. Eine Erweiterung um andere Funktionen stellt kein Problem dar, da hierzu nur die neuen Funktionsnamen und deren Ableitungsregeln in die Felder »Funktion« und »dFunk« einzutragen sind. Genaueres ist der Dokumentation des Listings zu entnehmen.

Das Listing stellt eine anschauliche, einfach gehaltene Lösung des Differenzierproblems dar. Da es keine Vereinfachungen (Optimierungen) durchführt, geraten selbst einfache Funktionen schnell zu aufgeblähten, unübersichtlichen Formeln. Auf der Diskette zum Heft finden Sie neben diesem Listing das Programm »How-To-Derive«, das neben der angesprochenen Vereinfachung eine weitere Besonderheit bietet: Es erklärt dem Benutzer Schritt für Schritt, wie die angegebene Funktion zu differenzieren ist. Dazu ist lediglich die gewünschte Funktion in das dafür vorgesehene String-Gadget einzugeben. Zudem unterstützt How-To-Derive 20 mathematische Funktionen und ist einfach zu bedienen. Wegen der detaillierten Erklärungen stellt es nicht nur eine Arbeitserleichterung dar, sondern ist auch als Lernprogramm zum Differenzieren geeignet. Viel Spaß beim Ableiten. *rz*

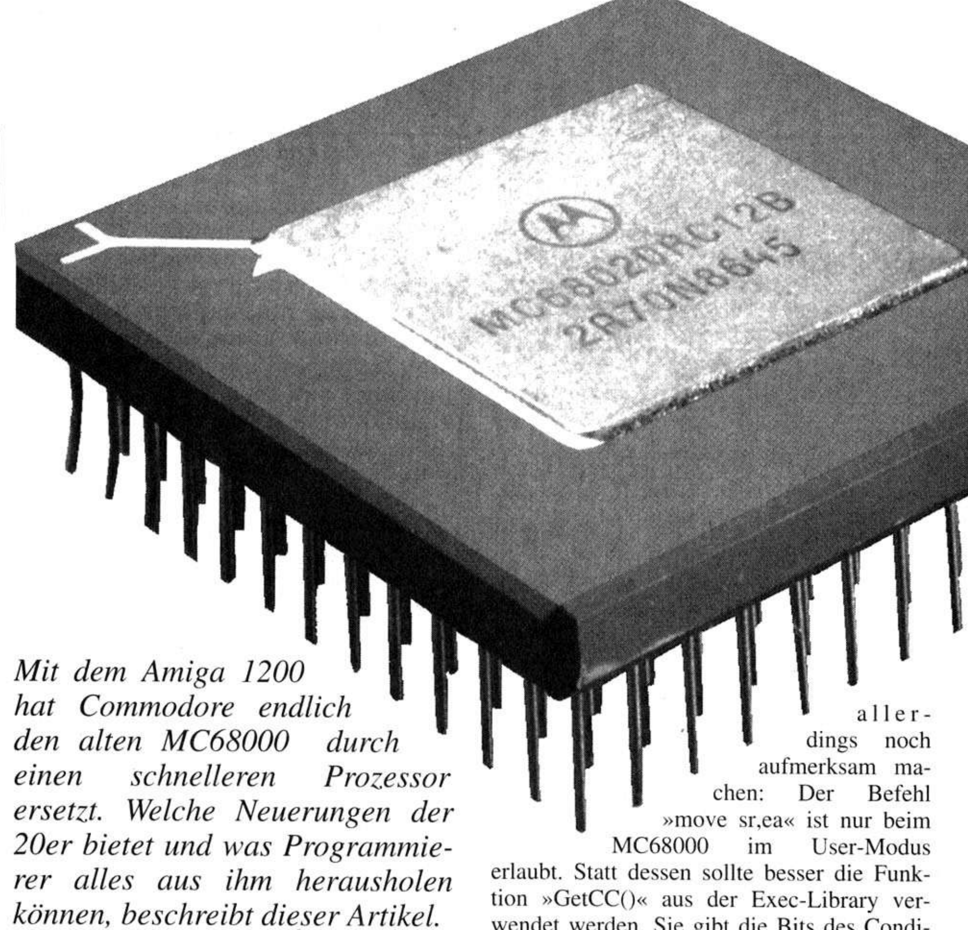
```
 1: /* Differenzieren mathemat. Ausdrücke */
 2: /* von Markus Öllinger */
 3:
 4: #include <ctype.h>
 5: #include <string.h>
 6: #include <stdio.h>
 7: #include <stdlib.h>
 8:
 9: typedef enum {OP_OPEN=-2,OP_CLOSE,OP_NON
    E=0,OP_ADD,OP_SUB,OP_MUL,OP_DIV,OP_NEG,O
    P_POT,OP_FUNC1} OP;
10:
11: struct Knoten {
12:  struct Knoten *l,*r;
13:  OP Operator;
14:  short int Pri;
15:  short int Len;
16:  char *Term;
17: };
18:
19: struct Knoten *Neuer_Operand(char *cp,ch
    ar **x);
20: void verwende_Operator(OP);
21: struct Knoten *Differenziere(struct Knot
    en *);
22:
23: extern char *Funktion[];
24:
25: OP operator_stack[1024];
                              /* Op.-Stapel */
26: int sp_operator;
                       /* Sein Stapelzeiger */
27: struct Knoten *operand_stack[1024];
28: int sp_operand;
29:
30: /* wandle Operatorenzeichen in Token */
31: OP Operator(int op)
32: {switch(op) {
33:   case '+': return OP_ADD;
34:   case '-': return OP_SUB;
35:   case '*': return OP_MUL;
36:   case '/': return OP_DIV;
37:   case '_': return OP_NEG;
38:   case '^': return OP_POT;
39:  }
40:  return 0;
41: }
42:
43: /* bestimme Priorität des Operators */
44: Pri(OP op)
45: {
46:  switch(op) {
47:   case OP_ADD:
48:   case OP_SUB: return 1;
49:   case OP_MUL:
50:   case OP_DIV: return 2;
51:   case OP_NEG: return 3;
52:   case OP_POT: return 4;
53:   case OP_NONE: return 0;
54:   default: return 5;
55:  }
56: }
57:
58: /* konstruiere Baum aus String */
59:
60: struct Knoten *Baum_aufbauen(char *term)
61: {char *cp=term;
62:  int i,l,pri,p;
63:  OP op,op2;
64:  char c;
65:
66:  while((c=*cp)!=0) {
67:   if(c=='(') operator_stack[sp_operator+
    +]=OP_OPEN;
68:   else if(isalnum(c)) {
69:    if(isalpha(c)) {
70:     for(i=0;Funktion[i];i++)
71:      if(!strncmp(Funktion[i],cp,l=strlen
    (Funktion[i]))) {
72:        pri=5;
73:        op=OP_FUNC1+i;
74:        cp+=l-1;
75:        goto operator;
76:      }
77:    }
```

```
78:    operand_stack[sp_operand++]=Neuer_Ope
     rand(cp,&cp);
79:    cp--;
80:   }
81:   else if(c==')')
82:    for(;;) {
83:     if(!sp_operator) goto error;
84:     op=operator_stack[--sp_operator];
85:     if(op!=OP_OPEN) verwende_Operator(op
     );
86:     else break;
87:    }
88:   else {
89:    op=Operator(c);
90:    pri=Pri(op);
91:
92:     /* Subtraktion oder Negation? */
93:
94:    if(op==OP_SUB && (cp==term || cp[-1]=
     ='(')) {
95:     op=OP_NEG;
96:     pri=3;
97:    }
98: operator: /* verwende alle Operatoren hö
     herer Priorität */
99:    while(sp_operator) {
100:    op2=operator_stack[sp_operator-1];
101:    p=Pri(op2);
102:
103:    /* Prioritätsvergleich. Achtung ^ ist
     rechtsassoziativ */
104:
105:    if(op2==OP_OPEN || p<pri || p==pri &
     & p==4)
106:     break;
107:    --sp_operator;
108:    verwende_Operator(op2);
109:   }
110:   operator_stack[sp_operator++]=op;
111:  }
112:  cp++;
113: }
114: while(sp_operator) {
115:  op2=operator_stack[--sp_operator];
116:  if(op2==OP_CLOSE) goto error;
117:  verwende_Operator(op2);
118: }
119: if(sp_operand!=1) goto error;
120: return operand_stack[--sp_operand];
121: error:
122:  return NULL;
123: }
124:
125: /* gibt Baum wieder frei (rekursiv) */
126: void Baum_abbauen(struct Knoten *k)
127: {if(k) {
128:   if(k->l) Baum_abbauen(k->l);
129:   if(k->r) Baum_abbauen(k->r);
130:   free(k);
131:  }
132: }
133:
134:  /* Hilfsroutinen */
135: struct Knoten *Neuer_Knoten(void)
136: {struct Knoten *k=malloc(sizeof(struct K
     noten));
137:  if(k==NULL) {
138:   fprintf(stderr,"Speicher voll!\n");
139:   exit(10);
140:  }
141:  memset((void *)k,0,sizeof(*k));
142:  return k;
143: }
144:
145: struct Knoten *Neuer_Operand(char *cp,ch
     ar **x)
146: {struct Knoten *k;
147:  char *s;
148:  k=Neuer_Knoten();
149:  s=cp;
150:  while(*++s=='.' || isalnum(*s));
151:  k->Term=cp;
152:  k->Len=s-cp;
153:  *x=s;
154:  return k;
155: }
156:
157: char *Op(OP op)
158: {static char c[]={" +-*/-^"};
159:  if(op>=OP_FUNC1) return Funktion[op-OP_
     FUNC1];
160:  return &c[op];
161: }
162:
163: struct Knoten *Neuer_Operator(OP op)
164: {struct Knoten *k;
165:  k=Neuer_Knoten();
166:  k->Operator=op;
167:  k->Pri=Pri(op);
168:  k->Term=Op(op);
169:  k->Len=op>=OP_FUNC1?strlen(k->Term):1;
170:  return k;
171: }
172:
173: /* holt ein o. zwei Operanden vom Stapel
     ,
174:  * verknüpft sie mit angeg. Operator und
175:  * legt Ergebnis wieder zurück. */
176:
177: void verwende_Operator(OP op)
178: {struct Knoten *k;
179:  int p=Pri(op);
180:
181:  k=Neuer_Operator(op);
182:  k->r=operand_stack[--sp_operand];
183:  switch(p) {
184:   case 1:
185:   case 2:
186:   case 4:
187:    k->l=operand_stack[--sp_operand];
188:    k->Operator=op;
189:    k->Pri=p;
190:    break;
191:  }
192:  operand_stack[sp_operand++]=k;
193: }
194:
195: /* erzeugt eine Kopie des Baums */
196: struct Knoten *BaumKopie(struct Knoten *
     k)
197: {struct Knoten *n=NULL;
198:  if(k) {
199:   n=Neuer_Operator(OP_NONE);
200:   *n=*k;
201:   n->l=BaumKopie(k->l);
202:   n->r=BaumKopie(k->r);
203:  }
204:  return n;
205: }
206:
207: /* Geht die in "Regel" gespeicherte Diff
     erenzierregel durch.
208:  * Diese ist in Präfixform gespeichert u
     nd hat folgende Syntax:
209:  * u entspricht dem linken Operanden (ni
     cht differenziert)
210:  * a entspricht dem linken Operanden (ab
     er differenziert)
211:  * v und b dasselbe rechts
212:  * + - * / ^ entsprechen den jeweilgen O
     peratoren.
213:  * _ entspricht dem (unären) Negationsop
     erator (Minus-Vorzeichen)
214:  * #XX entspricht einer Funktion, XX gib
     t die Funktionsnummer an.
215:  * #01 ist die Funktion mit dem im Felde
     lement Funktion[0]
216:  * gespeicherten Namen (hier ln), #02 en
     tspr. Funktion[1] usw.
217:  * Eine Ziffer steht für die angegebene
     Zahl.
218:  *
219:  * in u, v, a und b werden die
220:  * gleichnamigen Teilbäume erwartet. */
221:
222: char *Regel; /*zu verwendende Regel*/
223: struct Knoten *u,*v,*a,*b;
224:
225: /* Hier sind sie: die Differenzierregeln
      für
226:  * + - * / Negation und ^. */
227:
228: char *dRegeln[]={
229:  NULL,"+ab","-ab","+*av*bu","/-*av*bu^v2
     ",
230:  "_b","*^uv+*b#01u*/vua"};
231:
232: /* Die Funktionsnamen und die Regeln für
      die Ableitung */
233: char *Funktion[]={"ln","sinh","cosh","si
     n","cos",
234:  "exp",NULL};
235: char *dFunk[]={"/bv","*#03vb","*#02vb","
     *#05vb","*_#04vb",
236:  "*#06vb"};
237:
238: struct Knoten *Regeln(void)
239: {struct Knoten *k;
240:  int d;
241:
242:  switch(d=*Regel++) {
243:   case 'a': k=BaumKopie(a);break;
244:   case 'b': k=BaumKopie(b);break;
245:   case 'u': k=BaumKopie(u);break;
246:   case 'v': k=BaumKopie(v);break;
247:   case '_': k=Neuer_Operator(OP_NEG);k->
     r=Regeln();break;
248:   case '#': /* Funktion */
249:    sscanf(Regel,"%d",&d);
250:    Regel+=2;
251:    k=Neuer_Operator(d+OP_FUNC1-1);
252:    k->r=Regeln();
253:    break;
254:   default:
255:    if(isdigit(d)) { /* Zahl */
256:     k=Neuer_Knoten();
257:     k->Term=Regel-1;
258:     k->Len=1;
259:    } else {  /* Operator */
260:     k=Neuer_Operator(Operator(d));
261:     k->l=Regeln();
262:     k->r=Regeln();
263:    }
264:  }
265:  return k;
266: }
267:
268: /* erledigt das Differenzieren mittels R
     ekursion */
269: struct Knoten *Differenziere(struct Knot
     en *k)
270: {static char One='1',Zero='0';
271:  struct Knoten *d;
272:
273:  if(k->Operator==OP_NONE) {  /* Blatt (O
     perand) */
274:   d=Neuer_Knoten();
275:   if(k->Len==1 && k->Term[0]=='x')
276:    d->Term=&One;
277:   else d->Term=&Zero;
278:   d->Len=1;
279:   return d;
280:  }
281:  if(k->Operator>=OP_FUNC1) { /* Funktion
     */
282:   b=Differenziere(k->r);
283:   v=k->r;
284:   Regel=dFunk[k->Operator-OP_FUNC1];
285:   d=Regeln();
286:   Baum_abbauen(b);
287:   return d;
288:  }
289:
290:  /* Operator */
291:  if(k->l) d=Differenziere(k->l);
292:  else d=NULL;
293:  b=Differenziere(k->r);
294:  a=d;
295:  u=BaumKopie(k->l);
296:  v=BaumKopie(k->r);
297:  Regel=dRegeln[k->Operator];
298:  d=Regeln();
299:  Baum_abbauen(u);
300:  Baum_abbauen(v);
301:  Baum_abbauen(a);
302:  Baum_abbauen(b);
303:  return d;
304: }
305:
306: /* Baum zurück in String umwandeln */
307: /* hat der Vater v einen Sohn s mit eine
     m Operator
308:  * niedrigerer Priorität? */
309:
310: pri_klammer(struct Knoten *v,struct Knot
     en *s,char **sp)
311: {if(v->Pri>s->Pri && s->Pri!=0 || v->Pri
     ==s->Pri && v->Pri==4) {
312:    (*sp)[0]='(';
313:    (*sp)++;
314:    return 1;
315:  }
316:  return 0;
317: }
318:
319: /* mach ev. Klammer wieder zu */
320: void klammer_zu(char **sp,int kl)
321: {if(kl) {
322:    (*sp)[0]=')';
323:    (*sp)++;
324:  }
325: }
326:
327: void Baum_zu_String(struct Knoten *k,cha
     r *s)
328: {extern char *Baum_String(struct Knoten
     *,char *);
329:  *Baum_String(k,s)=0;
330: }
331:
332: char *Baum_String(struct Knoten *k,char
```



```
     *s)
333: {int kl;
334:
335:  if(k->Operator==OP_NONE) {
336:   strncpy(s,k->Term,k->Len);
337:   s+=k->Len;
338:   return s;
339:  }
340:  if(k->Operator>=OP_FUNC1) {
341:   strncpy(s,k->Term,k->Len);
342:   s+=k->Len;
343:   kl=pri_klammer(k,k->r,&s);
344:   if(!kl) *s++=' ';
345:   s=Baum_String(k->r,s);
346:   klammer_zu(&s,kl);
347:   return s;
348:  }
349:  if(k->Operator==OP_NEG) {
350:   *s++='-';
351:   kl=pri_klammer(k,k->r,&s);
352:   s=Baum_String(k->r,s);
353:   klammer_zu(&s,kl);
354:   return s;
355:  }
356:  kl=pri_klammer(k,k->l,&s);
357:  if(!kl && k->Operator==OP_POT && k->l->
    Operator==OP_POT) {
358:   *s++='(';
359:   kl=1;
360:  }
361:  s=Baum_String(k->l,s);
362:  klammer_zu(&s,kl);
363:  *s++=*Op(k->Operator);
364:  kl=pri_klammer(k,k->r,&s);
365:  if(!kl && (k->Operator==OP_SUB || k->Op
    erator==OP_DIV) && k->r->Pri==k->Pri) {
366:   *s++='(';
367:   kl=1;
368:  }
369:  s=Baum_String(k->r,s);
370:  klammer_zu(&s,kl);
371:  return s;
372: }
373:
374:  /*** Einsprung zum Differenzieren
375:   *** (Term muß korrekt sein)  ***/
376:
377: void Diff(char *Angabe,char *Ergebnis)
378: {struct Knoten *Root,*d;
379:
380:  Root=Baum_aufbauen(Angabe);
381:  Baum_zu_String(Root,Ergebnis);
382:  puts(Ergebnis);
383:  d=Differenziere(Root);
384:  Baum_zu_String(d,Ergebnis);
385:  Baum_abbauen(d);
386:  Baum_abbauen(Root);
387: }
388:
389: /* nur ein kurzer Check */
390: Term_korrekt(char *term)
391: {char c,*lastop=NULL,*p=term,*cp=term-1;
392:  int kl=0;
393:
394:  while((c=*++cp)!=0) if(!isspace(c)) {
395:   if(strchr("()+-*/^",c)) {
396:    if(c=='(') kl++;
397:    else if(c==')') {
398:     if(--kl<0 || lastop+1==cp || cp[-1]=
    ='(') return 0;
399:    }
400:    else {
401:     if(lastop+1==cp || c!='-' && cp[-1]=
    ='(') return 0;
402:     lastop=cp;
403:    }
404:   }
405:   else if(!isalnum(c) && c!='.') return
    0;
406:   *p++=c;
407:  }
408:  *p=0;
409:  return kl==0;
410: }
411:
412: char Angabe[1024],Ergebnis[1024];
413:
414: void main(int argc,char **argv)
415: {if(argc==2 && Term_korrekt(strcpy(Angab
    e,argv[1]))) {
416:   Diff(Angabe,Ergebnis);
417:   puts(Ergebnis);
418:  }
419: }                         © 1993 M&T
```

»Diff.c«: Das C-Programm leitet einen mathematischen Ausdruck ab



Mit dem Amiga 1200 hat Commodore endlich den alten MC68000 durch einen schnelleren Prozessor ersetzt. Welche Neuerungen der 20er bietet und was Programmierer alles aus ihm herausholen können, beschreibt dieser Artikel.

von Georg Herbold, Alexander Kochann und Oliver Reiff

Mit den beiden brandneuen Amigas, dem Amiga 1200 und dem Amiga 4000, geht die Ära des guten alten 68000er auf dem Amiga zu Ende. Im neuen Amiga 1200 gibt nun ein mit 14 MHz getakteter 68020er den Ton an. Der Amiga 4000 hingegen besitzt keinen fest eingebauten Prozessor. Hier kann eine beliebige Prozessorkarte eingesetzt werden, die mindestens einen 20er enthalten muß, wenn nicht gar einen 30- oder 40er. Ein alter 68000er bekäme mit den neuen Grafikauflösungen so seine Probleme.

Wir wollen uns hier auf den MC68020 beschränken, weil er nun den Grundbaustein bildet, auf dem künftig alle Programme aufbauen können – und sollten –, welche die AA-Chips voraussetzen. Wenn Sie also die neuen AA-Chips voll ausnutzen wollen, können Sie mindestens einen MC68020 als Herz des Systems voraussetzen, dennoch sollten Sie zu Beginn eines Programms testen, ob nicht etwa ein 68000er oder 68010er eingebaut ist. Wie man das macht, zeigt Programm 1, das zu »LessThan20« verzweigt, wenn eine ältere CPU installiert ist.

Da die MC68000er Familie aufwärtskompatibel ist, läuft die 20er Software auch auf einem höheren Prozessor. Die einzigen Einschränkungen stellen die Programme im Supervisor-Modus dar, in dem Anwenderprogramme aber sowieso nicht laufen sollten. Wer dennoch den Supervisor-Modus nutzen möchte, sollte sich entsprechende Literatur besorgen, da wir hier nicht darauf eingehen. Auf eine Inkompatibilität wollen wir Sie allerdings noch aufmerksam machen: Der Befehl »move sr,ea« ist nur beim MC68000 im User-Modus erlaubt. Statt dessen sollte besser die Funktion »GetCC()« aus der Exec-Library verwendet werden. Sie gibt die Bits des Conditioncode-Registers in d0 zurück und das, egal, welche CPU vorhanden ist.

Was ist beim MC68020 eigentlich anders als beim alten 68000er? Erst mal ist der 68020er natürlich schneller als sein Vorfahr. Das liegt an der höheren Leistung, der höheren Taktfrequenz und nicht zuletzt an dem 32 Bit breiten internen Datenbus. Zusätzlich ist ein 256 Byte großer Cache-Speicher im Prozessor vorhanden.

Cache mich: 68020 mit Nachbrenner

Beim Cache handelt es sich um einen extrem schnellen Speicher, in dem die letzten Befehle gespeichert und von dort wieder schnell abgerufen werden können. Dies macht sich besonders bei der Ausführung von Programmschleifen bemerkbar, die kleiner als 256 Byte sind, da diese komplett im Cache stehen und nicht mehr aus dem RAM geholt werden müssen. Dazu kommt noch das sog. Prefetch der Befehle. Dabei kann der 20er schon den nächsten Befehl aus dem Speicher, bzw. Cache holen und decodieren, während er einen anderen bearbeitet. Das einzige, was der Programmierer beachten sollte, ist die alte Regel, keinen selbstmodifizierenden Code zu schreiben, also nicht schreibend auf die unmittelbar abgearbeiteten oder folgenden Befehle zuzugreifen.

Der Rest geschieht ohne das Zutun des Programmierers, weshalb wir hier nicht näher darauf eingehen wollen. Die neuen Register, die bereits der MC68010 bereitsstellte, wollen wir ebenfalls nur kurz vorstellen, da es sich um Register handelt, die nur vom Super-

Programmierung 68020er Prozessor

Das neue Herz des Amiga

visor-Modus aus angesprochen werden dürfen. Im einzelnen sind dies folgende Register:

- VBR - das Vector Base Register
- SFC - Source Function Code
- DFC - Destination Function Code

Hierzu kamen für den MC68020 ebenfalls drei neue Register, die aber auch nur im Supervisor-Modus korrekt angesprochen werden dürfen.

- A7" (MSP) - der Master StackPointer
- CACR - das CAche Control Register
- CAAR - das CAche Adress Register

Viel wichtiger für Assembler-Programmierer ist die Tatsache, daß mit dem MC68020 nun endlich auch Worte und Langworte an ungeraden Adressen liegen können und es deshalb weniger häugfig zum berühmten Guru 80000003 (ungerade Adresse) kommen kann. Ganz verbannt werden konnte dieser Guru jedoch noch nicht, da Befehle auch weiterhin unbedingt an geraden Speicherstellen beginnen müssen. Auch Datenworte und -langworte sollten immer noch an geraden, bzw. durch vier teilbaren Adressen stehen, da sie dann auf einmal gelesen werden können. Im anderen Fall müssen sie sozusagen zerstückelt gelesen oder geschrieben werden, was natürlich zusätzliche Zeit in Anspruch nimmt. Achten Sie deshalb vor allem bei zeitkritischen Routinen auf die Position ihrer Worte und Langworte.

Doch auch auf diese Neuerung muß sich der Programmierer nicht umstellen. Viel interessanter für ihn sind die erweiterten

```
   move.l  4.w,a6
   btst    #1,296(a6)  * AttnFlags testen
   beq     LessThan20

   MC68020  * 20er-Assemblierung
            * einschalten

   ...

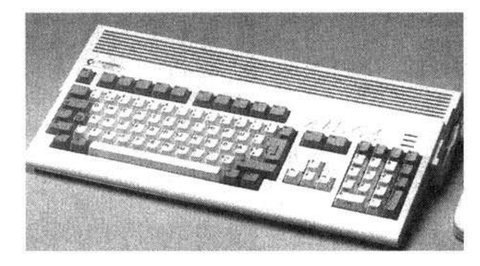
LessThan20
   MC68000               * 20er aus !

   Fehlerroutine
   bra Ende

   MC68020               * 20er wieder ein !

Ende
   ...                 © 1993 M&T
```

Listing 1: MC68020er-Test, hiermit testen Sie, was im Amiga steckt



Amiga 1200: In seinem Innern schlägt statt des 68000 ein 68020-Prozessor

Adressierungsarten. Einen Überblick der neuen Adressierungsarten finden Sie in Tabelle 1. Diese Tabelle erweckt zuerst den Anschein, als gäbe es nur sechs neue Adressierungsarten. Dies ist jedoch nur bedingt richtig. Durch optionale Angabe von Parametern ergeben sich daraus knapp 40 Möglichkeiten, Befehle zu adressieren.

Die ersten beiden Adressierungsarten stellen keine prinzipielle Neuerung dar. Sie sind lediglich Erweiterungen altbekannter Arten. Neu ist nur, daß alle Offsets jetzt 32 Bit breit sind und eine Skalierung angegeben werden kann. Die erste Adressierungsart heißt zu gut deutsch: Adreßregister indirekt mit Index mit oder ohne 32-Bit-Adreßdistanz und Skalierung.

Am besten vergleichen wir diese Art mit dem uns bekannten »d(An,Ri.s)«'. Der Befehl
`move.b #7,10(a4,d5.1)`
schreibt eine 7 an die Adresse, die sich aus 10+a4+d5 ergibt. Nichts anderes bewirkt der Befehl
`move.b #7,(10,a4,d5.1*1)`

Es gibt jedoch zwei Unterschiede:

Der Offset, der addiert wird, ist jetzt 32 statt 8 Bit groß. Dazu kommt die Skalierung. Sie bewirkt eine Multiplikation des Registerinhalts mit 1, 2, 4 oder 8, so daß sich bei
`move.b #7,(444444,a4,d5.1*4)`
die effektive Adresse sich wie folgt errechnet:
`ea = 444444+a4+d5*4.`

Die zweite neue Adressierungsart erweitert quasi das bekannte d16(pc,Ri.s) und entspricht der soeben erläuterten Art mit dem Unterschied, daß das erste Adreßregister durch den Programmcounter ersetzt wird. Auch hier bestehen die beiden Neuerungen darin, daß der Offset jetzt 32 Bit breit sein kann und daß der Registerinhalt mit 1, 2, 4 oder 8 multipliziert werden kann.

Das waren die sogenannten »einfachen Adressierungsarten«, die auch schon der MC68010 beherrschte. Mit dem 20er beginnt das Zeitalter der »komplexen Adressierungsarten« auf dem Amiga. Doch keine Angst: Auch wenn sie zunächst unübersichtlich erscheinen, sind sie relativ leicht zu erlernen. Ihnen zugrunde liegt das Prinzip der doppelt indirekten Adressierung. Bei der direkten Adressierung wird eine Speicherstelle direkt angegeben, bei der indirekten Methode ein Adreßregister, das die Speicherstelle enthält, und bei der doppelt indirekten Art ein Register, das eine Speicherstelle enthält, die wiederum auf die eigentliche Adresse zeigt. Alles klar?

Mnemonik	Art
(d1,An,Ri.s*scl)	ARI. [+Offset32]. [+Skalierung]
(label,pc,Ri.s*scl)	pc-relatic [+Offset32] [+Skalierung]
([d1,An],Ri.s*scl,d2)	doppelt indirekt, Post-Index
([d1,pc],Ri.s*scl,d2)	doppelt indirekt, Post-index, pc-relativ
([d1,An,Ri.s*scl],d2)	doppelt indirekt, Pre-Index
([d1,pc,Ri.s*scl],d2)	doppelt indirekt, Pre-Index, pc-relativ

Erklärungen zu Mnemonik:

d1 / d2	32-Bit-Offset
An	beliebiges Adressregister
Ri.s	beliebigers Register mit Größenangabe (word oder long)
scl	Skalierung (multipliziert Registerinhalt mit 1,2,4 oder 8)
pc	Programm Counter
ARI	AdreßRegister Indirekt (entspricht z.B. (a0))

Angaben bei Art in eckigen Klammern sind optional, können also weggelassen werden. Bei den letzten vier Adressierungsarten unter Mnemonik sind praktisch alle Bestandteile optional »([])« ist erlaubt

Tabelle 1: Die neuen Adressierungsarten in Verbindung mit dem 68020

Dazu zunächst ein ganz einfaches Beispiel, das bekanntlich mehr als 1000 Worte sagt: Der Befehl
`move.b #7,([a0])`
faßt quasi die Befehle
`move.l (a0),a0`
und
`move.b #7,(a0)`
zusammen, ohne jedoch a0 zu verändern!

Im nächsten Beispiel wollen wir es einmal so richtig kompliziert machen: statt des Befehls
`move.b #7,([10,a0],d5.w*8,20)`
könnte man auch
`move.l 10(a0),a0'`
`mulu #8,d5`
`move.b #7,20(a0,d5.1)`
schreiben, wobei allerdings a0 und d5 verändert würden.

Das Prinzip der Adressierung ist eigentlich ganz einfach: In eckigen Klammern steht die Speicherstelle, die doppelt indirekt adressiert wird. Danach stehende Angaben werden auch erst danach addiert.

Wie man in der Tabelle ebenfalls erkennen kann, darf anstelle des ersten Adreßregisters auch wieder der Programmzähler verwendet werden, so daß diese Adressierungsarten auch pc-relativ verwendet werden können. Ein Beispiel hierfür wäre:

```
move.l d0,([label(pc)],10)
```

Doch nicht nur neue Adressierungsarten wurden implementiert. Auch neue Befehle stehen zur Verfügung. Diese haben wir ebenfalls für Sie im Überblick zusammengefaßt (siehe Tabelle 2). Da haben wir als erstes die Bit-Feld-Befehle. Wie der Name sagt, lassen sich damit nicht nur einzelne, sondern mehrere Bits gleichzeitig beeinflussen. Dazu muß neben der Adresse auch das erste Bit und die Größe des Bit-Felds angegeben werden

Statt langen Worten: Bit-Felder

Besonders einfach wird die Bit-Feld-Programmierung dadurch, daß man als Programmierer weder an Byte-, Word- noch Langwortgrenzen gebunden ist.

Der folgende kurze Befehl

```
bfclr d0,{7:4}
```

löscht von Bit 7 an 4 Bit und funktioniert wie »bclr«, d.h. die angegebenen Bits werden gelöscht und die Flags werden gesetzt. Da die Befehle »bfchg«, »bfclr«, »bfset« und »bftst« genauso arbeiten wie ihre Ein-Bit-Kollegen, bedürfen sie keiner längeren Erklärung. Interessanter sind wohl die Befehle 'bfexts', bzw. »bfextu«. Sie übertragen ein Bit-Feld in ein Datenregister, das bei »bfextu« vorher gelöscht und bei »bfexts« wie bei »ext.x« vorzeichenrichtig erweitert wird. Umgekehrt schreibt »bfins« ein Bit-Feld aus einem Datenregister zurück in den Speicher. Als letzter Bit-Feld-Befehl bleibt »bfffo«, was die Abkürzung für »bit field – find first one« bedeutet. Er sucht die erste »1« in einem Bit-Feld und schreibt die Nummer dieses Bits ins Register dn. Findet er in dem angegebenen Bit-Feld keine »1«, wird die Länge des Bit-Felds plus eins zurückgegeben. Außerdem wird das Z-Flag entsprechend gesetzt.

```
        mc68020     * 20er einschalten
        lea         bcd,a0
        move.l      a0,a1
        lea         bcdende,a2
        move.l      a2,a3
        moveq       #3,d0
        move.l      d0,d1

loop    pack        -(a0),-(a2),#-$3030
                                * umwandeln
        dbra        d0,loop

loop2   unpk        -(a3),-(a1),#$3131
        * umwandeln und 1 mehr addieren
        dbra        d1,loop2
        rts

bcd     dc.l        0
bcdende
text    dc.b        "12345678"
                © 1993 M&T
```

Listing 2 : Demo zu den neuen Befehlen des 68020ers »pack« und »unpk«

Die nächste Gruppe bilden die Divisions- bzw. Multiplikationsbefehle. Der Befehl »divs.l« teilt jeweils zwei 32-Bit-Zahlen und speichert den Quotienten im 32-Bit-Zielregister ab. Ähnlich arbeitet »muls.l«, der zwei 32-Bit-Zahlen miteinander zu einer 32-Bit-Zahl multiplizieren. Selbstverständlich gilt gleiches für die vorzeichenlose Division, bzw. Multiplikation, also »divu« und »mulu«. Nichts Neues, werden Sie denken, was im Prinzip richtig ist. Aber diese Befehle wurden alle auf 32 Bit erweitert, wodurch sich z.T. ein anderer Syntax ergibt, wie die folgende Gruppe der Divisions- und Multiplikations-Befehle zeigt:

Bei »divs.l« können auch zwei Register als Ziel angegeben werden, beispielsweise dividiert »divs.l #7,d1:d0« das Register d0 durch 7. D1 wird dabei nicht beachtet. Erst nach der Division enthält das Datenregister d1 den 32 Bit großen Rest und d0 den 32 Bit großen Quotienten der Division.

Doch es gibt noch eine weitere Variante: Mit »divsl.l« kann sogar eine 64-Bit-Zahl dividiert werden. So würde »divsl.l #7,d1:d0« eine 64-Bit-Zahl, deren oberes Langwort in d1 und deren untereres Langwort in d0 untergebracht ist, durch 7 teilen. Die Rückgaberegister verhalten sich wie bei »divs.l«. Auch bei der Multiplikation zweier 32-Bit-Zahlen werden meistens 64-Bit-Zahlen entstehen, die dann ebenfalls in zwei Datenregistern untergebracht werden müssen. Dazu gibt es den Befehl »muls.l«, der das obere Langwort in »dr« und das untere Langwort in »dq« ablegt.

Die Anweisungen »pack« und »unpk« helfen dem Programmierer beim Umgang mit BCD-Zahlen. »Pack« wandelt dabei einen Zahlenstring in eine BCD-Zahl um. Dazu kann außerdem eine Konstante angegeben werden, die auf den String addiert wird. Es empfiehlt sich #-$3030. Bei dem entgegengesetzten Befehl »unpk« sollten wieder #$3030 addiert werden. Am besten betrachten Sie dazu das Programm 2. Es enthält nach einem Durchlauf in »text« den String »23456789« und in »bcd« die Zeichenfolge »$12345678«.

Das waren die wesentlichen Befehlsgruppen. Im folgenden Abschnitt wollen wir auf einige einzelne Befehle eingehen.

Da wäre beispielsweise als erstes der Befehl »cmp2.x grenzen,ea«. Mit ihm kann der Wert in ea gleichzeitig gegen zwei Grenzen aus einer Tabelle getestet werden. Dazu folgendes kurzes Beispiel, das testet, ob d0 zwischen 10 und 20 liegt und dementsprechend verzweigt:

```
cmp2.l grenzen,d0
bcs außerhalb
bcc innerhalb grenzen:
dc.l 10,20
```

BFCHG <ea>{bo.:bfl.}	Bitfeld invertierten
BFCLR <ea>{bo:bfl}	Bitfeld löschen
BFEXTS<ea>{bo:bfl},dn	Bitfeld vorzeichenrichtig übertragen
BFEXTU<ea>{bo:bfl},dn	Bitfeld vorzeichenlos übertragen
BFFFO <ea>{bo:bfl},dn	Finde erste 1 im Bitfeld
BFINS dn,<ea>{bo:bfl}	Bitfeld aus dn in Speicher schreiben
BFSET <ea>{bo:bfl}	Bitfeld setzen
BFTST <ea>{bo:bfl}	Bitfeld testen
DIVS.L<ea>,dn	Division mit Vorzeichen (32 Bit÷32 Bit)
DIVU.L<ea>,dn	vorzeichenlose Division (32 Bit÷32 Bit)
MULS.L<ea>,dn	Multiplikation mit Vorz. (32 Bit•32 Bit)
MULU.L<ea>,dn	Multiplikation ohne Vorz. (32 Bit•32 Bit)
DIVS.L<ea>,dr:dq	Division mit Vorzeichen (64 Bit÷32 Bit)
DIVSL.L <ea>,dr:dq	Division mit Vorzeichen (32 Bit÷32 Bit)
DIVU.L<ea>,dr:dq	vorzeichenlose Division (64 Bit÷32 Bit)
DIVUL.L <ea>,dr:dq	vorzeichenlose Division (32 Bit÷32 Bit)
MULS.L<ea>,dr:dq	Multiplikation mit Vorz. (32 Bit•32 Bit)
MULU.L<ea>,dr:dq	Multiplikation ohne Vorz. (32 Bit•32 Bit)
PACK-(Am),-(An),#x	bilde gepackte Dezimalzahl und addiere #x
PACKDm,Dn,#x "	
UNPK-(Am),-(An),#x	bilde ungepackte Dezimalzahl und addiere #x
UNPKDm,Dn,#x "	
CHK2.xgrenzen,ea	wie chk, nur mit zwei Grenzen
CMP2.xgrenzen,ea	wie cmp, nur mit zwei Grenzen
EXTB.Ldn	erweitert dn.b direkt vorzeichenrichtig auf dn.l
RTD #x	wie rts, addiert aber vorher #x auf den Stapel
MOVEccr,ea	schreibe Conditioncoderegister nach ea
BKPT#x	Breakpoint Nummer #x setzen
CALLM #x,ea	Aufruf eines Moduls mit Deskriptor
CAS.x Dc,Du,ea	Wenn Dc=ea, dann Du->ea, sonst ea->Dc
CAS2.xDc1:Dc2,Du1: Du2,(Rn):(Rm)	wie CAS nur mit zwei Vergleichswerten ???
MOVEC.x Rc,Rn	Lese Supervisor-Register
MOVEC.x Rn,Rc	Lade Supervisor-Register
MOVES.x Rn,ea	Lade alternativen Adreßbereich
MOVES.x ea,Rn	Lese alternativen Adreßbereich
RTM Rn	Rückkehr aus einem Modul und Löschen des Deskriptors
TRAPccea	bedingter Sprung in eine Exception
	erstes Bit des Feldes (0-31 oder Datenregister
	Größe des Bitfelds in Bits (1-32)

Tabelle 2: Die zusätzlichen Befehle des 68020er Prozessors

Neu ist auch der »extb.l«-Befehl, der das Lowbyte eines Registers vorzeichenrichtig auf das ganze Register erweitert. Er faßt also die Anweisungen »ext.w« und »ext.l« in einem Schritt zusammen. Ebenfalls eine Kurzfassung zweier Befehle bietet die Anweisung »rtd #x«, die »add.l #x,sp« und 'rts' in sich vereint. Diese Anweisung ist vor allem für die Programmierung von Unterroutinen interessant, die die Parameter auf dem Stack ablegen, wie dies bei Hochsprachen allgemein üblich ist.

Die anderen Befehle, die am Ende der Tabelle stehen, sollten Sie besser nicht verwenden, da sie teils privilegiert, teils nicht aufwärtskompatibel sind. Aber auch die erlaubten Befehle, die neuen Adressierungsarten und der schnellere Prozessor generell sollten Sie bereits zu neuen Höchstleistungen animieren. *ub*

Literaturhinweis:
Werner Hilf, »Mikroprozessoren für 32-Bit-Systeme«, Band 1, M&T-Verlag

Universeller Kreuzreferenzgenerator

Leichter durch den Dschungel

```
 7 #define Dim 4  // Dimension der Matrix
 8
 9 class Matrix   // Klassendeklaration
10 {
11   int matritze[Dim][Dim]; // durch Klasse
    gekapselt
12
13   public:      // folgende. Elemente von
    aussen zugreifbar
14
15   Matrix();    // Konstruktor initialisiert
    die Klasse
16
17   Matrix operator + (Matrix);
     // Ueberladen von +
18   void Ausgabe ();
19 };
20
21 Matrix::Matrix() // Klasse init.
22 {
23   int y;
24
25   for (int x = 0; x < Dim; x++)  // Zeilen
26     for (y = 0; y < Dim; y++)   // Spalten
27       matritze[x] [y] = Dim; // Alle
   Elemente sind DIM
28 }
29
30 Matrix Matrix::operator + (Matrix ma2)
31 {
32   Matrix ergebnis;
33   int y;
34
35   for (int x = 0; x < Dim; x++)
36    for (y = 0; y < Dim; y++)
37     ergebnis.matritze[x][y]
38     = matritze[x][y] + ma2.matritze[x][y];
39     return ergebnis;
40 }
41
42 void Matrix::Ausgabe ()
43 {
44   cout << "\n";
45   for (int x = 0; x < Dim; x++)
46     { for (int y = 0; y < Dim; y++)
47        cout << matritze[x][y] << ' ';
48        cout << "\n";
49     }
50   cout << "\n";
51 }
52
53 int main()
54 {
Kreuzreferenzen:
Ausgabe  18 42 58
cout     44 47 48 50
Dim       7 11 25 26 27 35 36 45 46
ergebnis 32 37 39
h         4  5
iostream  4
ma0      55 57
ma1      55 57
ma2      30 38 55 57 58
main     53
matritze 11 27 37 38 47
Matrix    9 15 17 21 30 32 42 55
stdlib   5x 25 27 35 37 38 45 47
y        23 26 27 33 36 37 38 46 47  © 1993 M&T
```

Listing 1: Ausschnitt aus einer Referenzliste, die mit dem Kreuzreferenzgenerator erstellt wurde. Der Ausschnitt zeigt, daß der einfache Ansatz nicht zwischen globalen und lokalen Variablen unterscheidet. So kommen x und y in mehreren Funktionen vor. Wie wäre es, wenn Sie das Programm so ausbauten, daß es Lokalitäten berücksichtigt?

Wenn ein Programm über eine lange Zeit reift, geraten häufig die gut funktionierenden Teile aus dem Blickfeld. Spätestens bei den abschließenden Arbeiten wünscht man sich ein Tool, das bei der Analyse des Quelltextes hilft, um den Überblick zurückzugewinnen. Der hier vorgestellte und für beliebige Sprachen einsetzbare Kreuzreferenzgenerator ist hervorragend geeignet, Quelltextdschungel zu durchdringen.

von Edgar Meyzis

Bei der Bereinigung von Quelltexten stellen sich Fragen wie die nach »unvollendeten Baustellen«, Vereinbarungsorten von Konstanten und Variablen oder der Vergabe aussagekräftiger Namen. Diese und ähnliche Fragestellungen lassen sich durchaus schrittweise mit einem Editor bewältigen. Leichter geht es jedoch mit einem Kreuzreferenzgenerator (KRG).

Vogelperspektive

Was leistet er? – Der zu analysierende Quelltext wird mit Zeilennummern versehen und in eine Datei geschrieben. Es folgt eine alphabetisch geordnete, formatierte Liste sämtlicher Namen (Bezeichner), die im Programmtext verwendet wurden. Für jeden Namen ist die Nummer der Zeile vermerkt, in der er vorkommt, über **Namen und Zeilennummern** (cross reference) fällt die Orientierung im Quelltext leicht. So erzeugte **Kreuzreferenzen (KR)** können mit Texteditoren bearbeitet werden. Listing 1 zeigt ausschnittsweise KR für ein Programm in C++.

Ein KRG muß den Wortschatz der Programmiersprache des zu analysierenden Quelltextes kennen, um reservierte Wörter (Schlüsselwörter) zu filtern. Unser **KRG** ist **universell** ausgelegt und **lernfähig**. Über einfache Textdateien können ihm die reservierten Wörter der jeweiligen Sprache (z.B. class oder WHILE) vorgegeben werden. Anhand der Endung eines Dateinamens erkennt er die zu berücksichtigende Sprache. Der KRG kann in der derzeitigen Version nicht zwischen Cluster, Modula-2 und Oberon differenzieren, da die drei Sprachen mit denselben »Suffixes« arbeiten. So bleibt Raum für Verbesserungen durch unsere Leser.

Selbst wenn Sie keinen KRG benötigen, bieten unsere Listings anschauliche Beispiele, Texte zu scannen und zu parsen sowie binäre Bäume zu errichten und darin herumzuturnen. Vielleicht benötigen Sie auch nur ein Tool, um die Zeilen eines Programms zu numerieren. Dann ignorieren Sie einfach die ausgegebene KR.

Spezifikation und Entwurf

Das Programm »Kreuzref.mod« (Listing 5) ist in **Modula-2** (M2AMIGA 4.1) geschrieben und so kommentiert, daß es leicht in eine andere Sprache umgesetzt werden kann. Es wurde für das **Betriebssystem** (OS) der **Version 1.3** ausgelegt. Bei strikter Anlehnung an OS 2.0 wären einige Funktionen einfacher zu implementieren gewesen. Auch hätte es sich angeboten, den ASL-Requester einzusetzen.

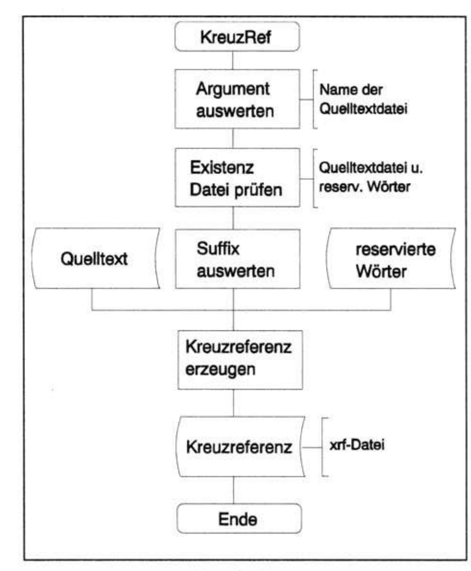


Bild 1: Datenfluß in unserem Programm Kreuzreferenz

Manchen Lesern wird das Thema schon von unserem Kurs **Mit System Entwickeln** [1] her bekannt sein. Er führte zum sprachunabhängigen Entwurf eines KRG. Dabei ergaben sich u.a. ein Diagramm für den Informationsfluß im KRG ähnlich Abb. 1 und die »Spielregeln« gem. Listing 2, um Quelltexte Programmiersprachen zuzuordnen. Wichtige Ergebnisse des Entwurfs (AMIGA 9/91, S. 166 ff) sollen ausschnittsweise und stark verkürzt wiederholt werden, damit Sie das Programm leichter nachvollziehen können.

Bäume pflanzen

Der KRG setzt dynamische Datenstrukturen ein, um Quelltexte mit beliebig vielen Namen (innerhalb der Grenzen des verfügbaren Arbeitsspeichers) analysieren zu können. Die Adressen der **Namen** und der **reservierten Wörter** werden getrennt in zwei sortierten, **binären Bäumen** abgelegt. Die Bäume lassen sich sehr effizient bearbeiten, wenn die einzelnen Knoten systematisch aufeinander verweisen. Die Verwaltung von Namen und reservierten Wörtern in einheitlich organisierten Bäumen bietet den Vorteil, mit **einem Satz an Algorithmen** für beide Bäume auszukommen.

```
CONST SprachID = "MOD_DEF_CPP_HPP_BAS_PAS_C_H";
                 (*^   ^   ^   ^   ^   ^   ^ ^ *)
                 (*0   4   8  12  16  20  24 26 *)
  CASE id OF
    0..4 :  wortText := "S:Modula.Words";
              (* zgl. Cluster*)
            sprache  := Modula; (* und Oberon *)
  | 8..12:  wortText := "S:CPP.Words";
            sprache  := Cpp;
  | 16 :    wortText := "S:Basic.Words";
            sprache  := Basic;
  | 20 :    wortText := "S:Pascal.Words";
            sprache  := Pascal;
  | 24..26 : wortText := "S:C.Words";
            sprache  := C;
  END;                          © 1993 M&T
```

Listing 2: Aus dem Suffix wird auf die Sprache des Quelltextes geschlossen. Modula-2 steht zugleich für Cluster und Oberon. Mit einem zusätzlichen Argument bei Aufruf des Programms ließe sich differenzierter vorgehen.

Für die Erzeugung von KR reicht das einfache Verfahren der Binärsuche [2] aus. Dazu sind die Knoten der Bäume entsprechend Abb. 2 anzulegen und z.B. gem. Listing 3 zu implementieren. Ein kleiner Unterschied in der Verwaltung der beiden Informationsarten (Name oder reserviertes Wort) besteht schon. Für reservierte Wörter ist »numList« (Anfang der Liste mit den Zeilennummern) »NIL«, da sie nicht in die KR eingehen. Im Baum der Namen hingegen verweist »numList«´ auf eine einfach verkettete Liste, deren Elemente eine Zeilennummer und einen Zeiger auf das nächste Element enthalten (vgl. Abb. 2). So ist sichergestellt, daß unser KRG mit beliebig häufigen Vorkommen eines Namens umgehen kann (Merkmal dynamischer Programmierung).

Links vor rechts

Die Verhältnisse im Baum selbst gehen aus Abb. 3 hervor. Es wird veranschaulicht, wie Knoten mit reservierten Wörtern oder Namen in Bäume einzufügen sind. Der **Suchvorgang** innerhalb eines Baums ist in Abb. 4 **als Struktogramm** dargestellt und in Listing 4 als Prozedur »NamenSuchen« implementiert.

Wenden wir doch einmal das Struktogramm auf den Baum in Abb. 3 an und begeben uns auf die **Suche** nach dem reservierten Wort **BYTE**:

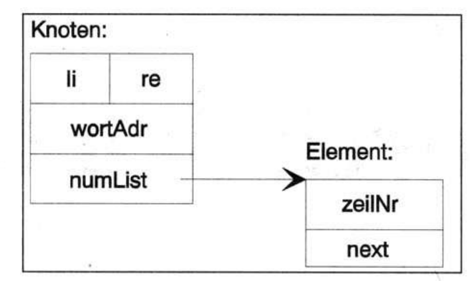


Bild 2: Die Knoten der Binärbäume enthalten insgesamt vier Adreßinformationen. Die Einträge li(nks) und re(chts) ermöglichen es, einen Baum aufzuspannen. wortAdr verweist auf ein reserviertes Wort oder auf einen Namen. Die Zeilennummern werden in einer einfach verketteten Liste ab num List festgehalten

Step 1: Der Wurzelknoten verweist auf Knoten 1 mit der Adresse von »ABS«. Der Vergleich von ABS mit BYTE ergibt, daß nach rechts weiter zu verzweigen ist: q := p^.re;).

```
Step 2: BYTE < DEC  somit:  q := p^.li;
Step 3: BYTE > BY   somit:  q := p^.re;
Step 4: BYTE = BYTE somit:  RETURN p;
```

Das Beispiel zeigt, wie einfach es ist, Informationen im Baum schnell zu finden.

Rekursiv von Ast zu Ast

Die Routine **NamenSuchen** (Listing 4) wurde **nicht rekursiv** implementiert, um den Einstieg in die Baumtechnik zu erleichtern. In der Prozedur **WeiterImBaum** (Listing 4) finden Sie ein **rekursives Verfahren**, um sich mit wenigen Programmzeilen von Ast zu Ast der beiden Bäume zu schwingen und die enthaltenen Informationen auszugeben. Es sollte gleichfalls leicht nachzuvollziehen sein. **Listing 4** enthält die **vollständigen Baumalgorithmen**.

```
TYPE
   ZeichenPtr = POINTER TO CHAR;
   ListPtr    = POINTER TO Element;

   Element   = RECORD
    (* Knoten einer einfach verketteten Liste *)
    zeilNum : INTEGER;
           (* Zeilennummer eines Namens *)
    next    : ListPtr; (* naechstes Element*)
   END;

  BaumPtr    = POINTER TO Knoten;

  Knoten     = RECORD
    (* Blaetter der beiden Binaerbaeume *)
    wortAdr : ZeichenPtr;
           (* Name oder reserv. Wort *)
    numList : ListPtr;
           (* auf Liste mit Zeilennr *)
    li, re  : BaumPtr;  (* Baum verzweigt *)
   END;                © 1993 M&T
```

Listing 3: Zwei Datenstrukturen formen die binären Bäume. Die einfach verkettete Liste setzt sich aus dem Typ Element zusammen, um Nummern der Zeilen aufzunehmen, in denen die Namen des analysierten Programms vorkommen.

Da die Speicheranforderungen über den »Heap« erfolgen, ist es nicht explizit erforderlich, die angeforderten Bereiche freizugeben. Bei Programmende übernimmt das Laufzeitsystem diese Aufgabe. Würde das Programm beim Aufruf mehr als eine Datei als Argument akzeptieren, dann wäre zumindest der Namensbaum abzubauen.

(bitt lesen Sie weiter auf Seite 100(

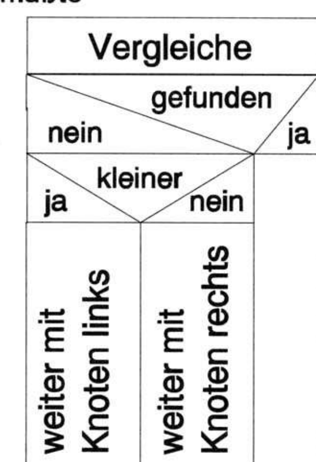


Bild 4: Die Suche nach einem Namen in einem binären Baum in Form eines Struktogramms dargestellt

```
165: * Hier sollten Sie alle auftretenden Fehler
166: * abfangen und dem Benutzer mit Hilfe von
167: * Requestern darauf hinweisen. Wir haben
168: * darauf verzichtet, um das Programm nicht
169: * unnötig zu verlängern.
170:
171: No_OS2     moveq     #-10,d7
172:            bra.s     Err_OS2
173: No_Screen  moveq     #-10,d7
174:            bra.s     Err_Screen
175: No_Window  moveq     #-10,d7
176:            bra.s     Err_Window
177: No_Menu    moveq     #-10,d7
178:            bra.s     Err_Menu
179: No_Layout  moveq     #-10,d7
180:            bra.s     Err_Layout
181: *--------------------
182: GetTextLenght
183:            movem.l   d1/a0-a2/a6,-(sp)
184:            move.l    a0,a2
185:            move.l    Screen,a1
186:            lea       84(a1),a1          * RastPort
187:            moveq     #-1,d0
188: .loop      addq.l    #1,d0
189:            tst.b     (a2)+
190:            bne.s     .loop
191:            move.l    GfxBase,a6
192:            jsr       -54(a6)            TextLenght
193:            movem.l   (sp)+,d1/a0-a2/a6
194:            rts
195: *--------------------
196: PrintText  movem.l   d0-d1/a0-a2/a6,-(sp)
197:            move.l    a0,a2
198:            move.l    RastPort,a1
199:            moveq     #-1,d0
200: .loop      addq.l    #1,d0
201:            tst.b     (a2)+
202:            bne.s     .loop
203:            move.l    GfxBase,a6
204:            jsr       -60(a6)            Text
205:            movem.l   (sp)+,d0-d1/a0-a2/a6
206:            rts
207: *--------------------
208: SetPen     movem.l   d0-d1/a0-a1/a6,-(sp)
209:            move.w    d2,d0
210:            move.l    RastPort,a1
211:            move.l    GfxBase,a6
212:            jsr       -342(a6)           SetAPen
213:            movem.l   (sp)+,d0-d1/a0-a1/a6
214:            rts
215: Move       movem.l   d0-d1/a0-a1/a6,-(sp)
216:            move.l    RastPort,a1
217:            move.l    GfxBase,a6
218:            jsr       -240(a6)           Move
219:            movem.l   (sp)+,d0-d1/a0-a1/a6
220:            rts
221:            opt       p-
222: IntBase              dc.l      0
223: GfxBase              dc.l      0
224: GadBase              dc.l      0
225: RastPort             dc.l      0
226: MessagePort          dc.l      0
227: DrawInfo             dc.l      0
228: VisualInfo           dc.l      0
229: Font                 dc.l      0
230: Menu                 dc.l      0
231: Window               dc.l      0
232: YSize                dc.w      0
233: BaseLine             dc.w      0
234: WindowData           dc.w      0,0,0,0,1
235:                      dc.l      $300,$2100f,0,0,WindowName
236: Screen               dc.l      0,0
237:                      dc.w      0,0,-1,-1,15
238: WindowTags           dc.l      TAG_USER+147,1
239: WA_InnerWidth        dc.l      TAG_USER+118,0
240: WA_InnerHeight       dc.l      TAG_USER+119,0
241:                      dc.l      0
242: NewMenu              dc.b      1,0
243:                      dc.l      MenuTitle_1,0
244:                      dc.w      0
245:                      dc.l      0,0
246:                      dc.b      2,0
247:                      dc.l      MenuPoint_1,T_Key
248:                      dc.w      $109
249:                      dc.l      0,0
250:                      dc.b      2,0
251:                      dc.l      -1,0
252:                      dc.w      0
253:                      dc.l      0,0
254:                      dc.b      2,0
255:                      dc.l      MenuPoint_3,V_Key
256:                      dc.w      0
257:                      dc.l      0,0
258:                      dc.l      0
259: NewMenuTags          dc.l      TAG_USER+$80043,1
260:                      dc.l      0
261: VersionString        dc.b      '$VER: WB-Beispielprogramm (11.11.92)',0
262: IntName              dc.b      'intuition.library',0
263: GfxName              dc.b      'graphics.library',0
264: GadName              dc.b      'gadtools.library',0
265: PupScreen            dc.b      'Workbench',0
266: WindowName           dc.b      'Testfenster',0
267: MenuTitle_1          dc.b      'Projekt',0
268 MenuPoint_1          dc.b      'Test',0
269: MenuPoint_3          dc.b      'Verlassen',0
270: Text_1               dc.b      'Nicht vergessen:',0
271: Text_2               dc.b      'Die WB ist unberechenbar...',0
272: T_Key                dc.b      'T',0
273: V_Key                dc.b      'V',0
274                      END             © 1993 M&
```

WB-Beispielprogramm: Das Programm zeigt, was unter OS 2.0 zu beachten ist (Fortsetzung von Seite 9)

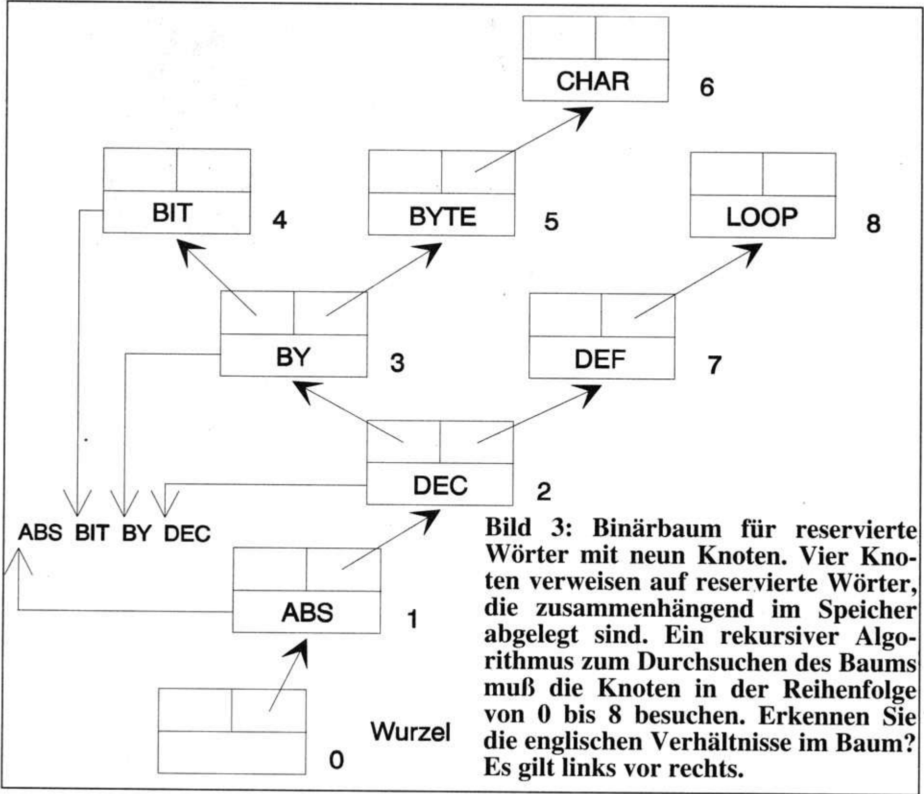


Bild 3: Binärbaum für reservierte Wörter mit neun Knoten. Vier Knoten verweisen auf reservierte Wörter, die zusammenhängend im Speicher abgelegt sind. Ein rekursiver Algorithmus zum Durchsuchen des Baums muß die Knoten in der Reihenfolge von 0 bis 8 besuchen. Erkennen Sie die englischen Verhältnisse im Baum? Es gilt links vor rechts.

Das Programm gliedert sich insgesamt in **fünf Module**, die über schmale Schnittstellen gekoppelt sind. Das Hauptmodul **KreuzRef.mod** arbeitet mit den Modulen **BinärBaum** (allgemeine Routinen für die Arbeit mit den Bäumen), **ResWortBaum** (Baum mit reservierten Wörtern aufspannen), **ReferenzBaum** (Namensbaum bilden, KR erzeugen) und **DateiHandling**. Listing 5 enthält die Schnittstellendefinitionen der Module, um Sie mit der Programmstruktur vertraut zu machen. Die **vollständigen Listings** finden Sie auf der zum Heft gehörenden **PD-Diskette** im Verzeichnis »KreuzReferenz« (siehe auch Seite 114).

Wann lichten Sie Ihre Urwälder?

Zur Funktion des Programms ist abschließend noch anzumerken, daß es bei erweiterter Selektion (SHIFT Klick Datei, SHIFT Klick KreuzRef) von der Workbench arbeitet, ohne jedoch ein Icon für die erzeugte Referenz anzulegen. Die Dateien mit den **reservierten Wörtern** müssen sich im **Verzeichnis S:** (vgl. Listing 2) befinden. Für Basic, C, C++ und Modula-2 sind entsprechende Dateien (als Muster) mit einem Grundstock an reservierten Wörtern auf der PD-Diskette beigefügt. Die Reihenfolge in den Dateien ist unerheblich, da ohnehin sortiert wird. *ub*

Literatur:

1] E. Meyzis, Mit System entwickeln, AMIGA 3/91, 5/91, 7/91, 9/91 und 12/91

2] N. Wirth, Algorithmen und Datenstrukturen, Stuttgart, 1983


```
  1: IMPLEMENTATION MODULE BinaerBaum;
  2: FROM Arts IMPORT  Assert;
  3: FROM DateiHandling IMPORT  GetSprachID, Spra
     che;
  4: FROM Storage IMPORT  ALLOCATE;
  5: FROM SYSTEM IMPORT  ADR;
  6: FROM Terminal IMPORT  Write, WriteLn;
  7: PROCEDURE BaumAnlegen(VAR wurzel : BaumPtr);
  8: BEGIN
  9:  ALLOCATE(wurzel, SIZE(Knoten));
 10:  Assert(wurzel # NIL, ADR("Baum nicht anlegb
     ar"));
 11:  wurzel^.re := NIL;
 12: END BaumAnlegen;
 13: PROCEDURE NamenSuchen(VAR p : BaumPtr; (* im
      Baum *)
 14:  namen    : ZeichenPtr;
 15:  isPascal : BOOLEAN;
 16:  VAR rel  : Relation) : BaumPtr;
 17: (* Die lokale Prozedur "Vergleiche" (zwei Ze
     ichenketten) vermeidet den rekursiven Aufruf
      von "NamenSuchen" mit dem Vorteil, keine la
     nge Parameterliste kopieren zu muessen. *)
 18: VAR  q : BaumPtr;
 19: PROCEDURE Vergleiche (neu, vglWort:ZeichenPt
     r): Relation;
 20: (* zwei Strings zeichenweise *)
 21: VAR
 22:  rel : Relation;
 23: BEGIN
 24:  rel:= gleich; (* vorlaeufige Annahme *)
 25:  LOOP
 26:   IF CAP(neu^) # CAP(vglWort^) THEN (* Zeich
     en ungleich? *)
 27:    EXIT (* Vergleich abbrechen *)
 28:   END;
 29:   IF neu^ <= " " THEN (* Ende der Kette? *)
 30:    RETURN rel (* Vergleichsergebnis *)
 31:   END;
 32:   IF NOT isPascal THEN
 33:    IF neu^ < vglWort^ THEN (* zeichenweiser
     Vergl. *)
 34:     rel:= kleiner;
 35:    ELSIF neu^ > vglWort^ THEN
 36:     rel:= groesser;
 37:    END;
 38:    ELSE (* fuer PASCAL alles GROSS *)
 39:     IF CAP(neu^) < CAP(vglWort^) THEN
 40:      rel:= kleiner;
 41:     ELSIF CAP(neu^) > CAP(vglWort^) THEN
 42:      rel:= groesser;
 43:     END;
 44:    END;
 45:    INC(neu); (* naechstes Zeichen im String*)
 46:    INC(vglWort); (*              "          *)
 47:   END; (* LOOP *)
 48:   IF CAP(neu^) > CAP(vglWort^) THEN
 49:     RETURN groesser
 50:   ELSE
 51:     RETURN kleiner
 52:   END;
 53: END Vergleiche;
 54: BEGIN
 55:  q:= p^.re; (* an Wurzel beginnen *)
 56:  rel:= groesser; (* vorlaeufig *)
 57:  WHILE q # NIL DO (* bis alle Knoten durchsu
     cht sind, falls vorher nicht gefunden. *)
 58:   p:= q;
 59:   rel := Vergleiche(namen, p^.wortAdr);
 60:   IF rel = gleich THEN
 61:    RETURN p  (* gefunden *)
 62:   ELSIF rel = kleiner THEN
 63:    q:= p^.li                     (* im linken Zw
     eig weiter *)
 64:   ELSE
 65:    q:= p^.re                    (* im rechten Zw
     eig weiter *)
 66:   END;
 67:  END;
 68:  RETURN NIL;
 69: END NamenSuchen;
 70: PROCEDURE KnotenEinfueg(VAR p : BaumPtr;
 71:          rel   : Relation;
 72:          namen : ZeichenPtr) : BaumPtr;
 73: VAR q : BaumPtr;
 74: BEGIN
 75:  ALLOCATE(q, SIZE(p^));  (* neuen Knoten *)
 76:  Assert(q # NIL, ADR("kein neuer Knoten"));
 77:  WITH q^ DO
 78:   wortAdr := namen;                (* Nutzinf
     o eintragen *)
 79:   numList := NIL;
 80:   li := NIL;
 81:   re := NIL;
 82:  END;
 83:  IF rel = kleiner THEN
 84:   p^.li := q (* Knoten anfuegen entspr. *)
 85:  ELSE        (* des Ordnungsschemas im  *)
 86:   p^.re := q (* Baum.                   *)
 87:  END;
 88:  RETURN q;
 89: END KnotenEinfueg;
 90: PROCEDURE ZeigeBaum(p : BaumPtr); (* Test *)
 91:                (* fuer beide Baeume *)
 92: PROCEDURE InfoAusgeben(p:BaumPtr);
 93: (* gespeicherte Namen bzw. reservierte Woert
     er *)
 94: VAR z : ZeichenPtr;
 95: BEGIN
 96:  z := p^.wortAdr;  (* Anfang der Zeichenkett
     e *)
 97:  WHILE z^ > " " DO  (* solange Ende  Kette u
     nerreicht *)
 98:    Write(z^);     (* zeichenweise ausgeben *)
 99:     INC(z);    (* auf das naechste Zeichen *)
100:  END;
101:  WriteLn;
102: END InfoAusgeben;
103: PROCEDURE WeiterImBaum(p : BaumPtr);
                              (* rekursiv *)
104: BEGIN
105:  IF p # NIL THEN   (* ein Knoten erreicht *
     )
106:   WeiterImBaum(p^.li);  (* soweit im linken
                            Zweig, wie es geht *)
107:   InfoAusgeben(p);    (* dann alle zwischenz
     eitlich durchlaufenen Knoten in umgekehrter
     Reihenfolge ausgeben *)
108:   WeiterImBaum(p^.re); (* ab in den rechten
      Zweig*)
109:  END;
110: END WeiterImBaum;
111: BEGIN
112:  WriteLn;
113:  WeiterImBaum(p^.re); (* der linke Zweig des
     Wurzelknotens ist unbenutzt. *)
114: END ZeigeBaum;
115: END BinaerBaum.               © 1993 M&T
```

Listing 4: Die Routinen zeigen, wie einfach binäre Bäume anzulegen, zu verwalten und auszugeben sind. Ähnliche Aufgaben werden durch wiederholten Aufruf eines Unterprogramms (NamenSuchen) und rekursiv (WeiterImBaum) erledigt, um die Einarbeitung in Baumalgorithmen zu erleichtern.

```
 1: MODULE KreuzRef;  (* Hauptmodul Kreuzreferenzgenerator *)
 2: FROM DateiHandling IMPORT DateienOeffnen, DateienSchliessen;
 3: FROM ReferenzBaum  IMPORT ReferenzErstellen;
 4: BEGIN
 5:   DateienOeffnen;
 6:   ReferenzErstellen;
 7: CLOSE (* immer ausfuehren, auch bei Abbruch *)
 8:    DateienSchliessen;
 9: END KreuzRef.
10: (* Das Hauptmodul arbeitet mit vier weiteren Modulen, deren Schnittste
    llen wie folgt definiert sind: *)
11: (* 1111111111 *)
12: DEFINITION MODULE DateiHandling;
13: FROM FileSystem IMPORT File;
14: TYPE Sprache = (Basic, C, Cpp, Modula, Pascal);
15:   (* um Sonderbehandlungen zu ermoeglichen. Cluster und Oberon firmiere
    n unter Modula-2 *)
16: VAR wortQuelle,  (* Datei mit den reservierten Woertern im Verzeichnis
      "S:", z.B. "C.Words" *)
17:   quelle, (* zu analysierender Quelltext, z.B. "Hallo.cpp" *)
18:   senke  : File;  (* nimmt Kreuzreferenz auf und hat denselben Pfad wie
      "quelle", z.B. "Hallo.xrf" *)
19: PROCEDURE FileExists(name : ARRAY OF CHAR) : BOOLEAN; (* Datei vorhand
    en ? *)
20: PROCEDURE GetSprachID() : Sprache;
21: (* ergibt sich aus dem Suffix des Dateinamens fuer den Quelltext *)
22: PROCEDURE DateienOeffnen;
23: (* alle drei Dateien; Programm bei Fehlern abbrechen *)
24: PROCEDURE DateienSchliessen;
25: (* die drei Dateien *)
26: END DateiHandling.
27: (* 222222222 *)
28: DEFINITION MODULE BinaerBaum;
29: TYPE
30:   ZeichenPtr = POINTER TO CHAR;
31:   Relation   = (kleiner, groesser, gleich); (* Ergebnis des alphabetis
    chen Vergleichs von Zeichenketten *)
32:   ListPtr    = POINTER TO Element;
33:   Element    = RECORD (* Knoten einer einfach verketteten Liste *)
34:   zeilNum : INTEGER; (* Zeilennummer eines Namens *)
35:   next     : ListPtr; (* naechstes Element*)
36:  END;
37:   BaumPtr    = POINTER TO Knoten;
38:   Knoten     = RECORD (* Blaetter der beiden Binaerbaeume *)
39:   wortAdr : ZeichenPtr; (* Name oder reserv. Wort *)
40:   numList : ListPtr;    (* auf Liste mit Zeilennr *)
41:   li, re  : BaumPtr;  (* Baum verzweigt *)
42:  END;
43: PROCEDURE BaumAnlegen(VAR wurzel : BaumPtr);
44: (* fuer Wurzelknoten des Baumes mit den reservierten Woertern bzw. auf
    Baum mit den Namen des zu analysierenden Programms *)
45: PROCEDURE NamenSuchen(VAR p : BaumPtr;
46:   namen    : ZeichenPtr;
47:   isPascal : BOOLEAN;
48:   VAR rel  : Relation) : BaumPtr;
49: (* in beiden Baeumen Ergebnis: NIL, wenn namen nicht gefunden, sonst A
    dresse des Knotens
50:   p: weist auf Knoten, der den gesuchten Namen enthalten muesste, fall
    s Ergebnis NIL
51:   namen: ist zu suchen
52:   isPascal: um Gross-/Kleinschreibung richtig zu handeln
53:   rel:      so ging der Vergleich aus   *)
54: PROCEDURE KnotenEinfueg(VAR p : BaumPtr;
55:   rel   : Relation;
56:   namen : ZeichenPtr) : BaumPtr; (* an der Stelle p im Baum *)
57: PROCEDURE ZeigeBaum(p : BaumPtr); (* eingebaute Testroutine *)
58: END BinaerBaum.
59: (* 333333333 *)
60: DEFINITION MODULE ResWortBaum;
61: FROM BinaerBaum IMPORT BaumPtr;
62: FROM FileSystem IMPORT File;
63: PROCEDURE WortBaumAnlegen(wortDatei : File) : BaumPtr;
64: (* fuer die reservierten Woerter einer Sprache *)
65: END ResWortBaum.
66: (* 444444444 *)
67: DEFINITION MODULE ReferenzBaum;
68: FROM BinaerBaum IMPORT BaumPtr;
69: PROCEDURE ReferenzErstellen;
70: (* spannt beide Baeume auf (res. Woerter u. Referenz), speichert das E
    rgebnis in eine Datei mit dem Namen der Quelltextdatei (im selben Pfad
    ) und dem Suffix ".xrf"*)
71: END ReferenzBaum.
72:
73:   © 1993 M&T
```

Listing 5: Unser Programm »Kreuzreferenzgenerator« besteht aus fünf Modulen, die recht lose gekoppelt sind. Das Modul »BinaerBaum« enthält die grundlegenden Routinen für die Arbeit mit binären Bäumen. Es wird von den Modulen »ResWortBaum« und »ReferenzBaum« eingesetzt.

3-State Computertechnik

MULTIVISION 500/2000

Kein Interlace-Flimmern mehr!

MV2000 wird in den Videoslot des A2000 B/C eingesteckt
MV500 findet Platz im Sockel des Videochips (Denise)

Test Kickstart 7'91: sehr gut! Auch für Amiga 500plus!

- Volles Overscan (768 x 598 Punkte), 4096 Farben
- 50 Hz Vollbildfrequenz, per Software (im Lieferumfang) bis 100 Hz einstellbar
- Double-Scan-Modus, die schwarzen Zwischenzeilen verschwinden
- Integrierter Stereo-Audio-Verstärker
- kompatibel mit jeder Software
- VGA-kompatibler Videoausgang zum Anschluß von VGA/Multiscan-Monitoren

Die Leser des Amiga-Magazins wählten MultiVision zum Produkt des Jahres 1991 & 1992.



FLOPPY DRIVE 3,5"

Bus bis df3 · abschaltbar · extern für alle Amigas · mit Metallgehäuse
Made in Germany

149,-

CHIP 2 MB

Adapter-Platine für A500 & A2000 B/C/D
Erweitert das ChipRAM von 1 MB auf 2 MB
Einfach einstecken, Einbau ohne Löten
inkl. 1 MB RAM
und 8375 Super-Agnus

349,-

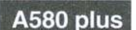
A580 / A580 plus

A580

Speichererweiterung für A500 intern auf 2.3 MB

inkl. Uhr & Akku
& Gary-Ad

249,-



1.0 MB ChipRAM &
2.5 MB Gesamtspeicher
inkl. CPU-Adapter

299,-



MegaMix 500/2000

2.0 MB bis 8.0 MB FastRAM-Erweiterung für A500 & A2000 • null Waitstates • autokonfigurierend • abschaltbar • für A500 extern im formschönen Gehäuse mit durchgeführtem Systembus

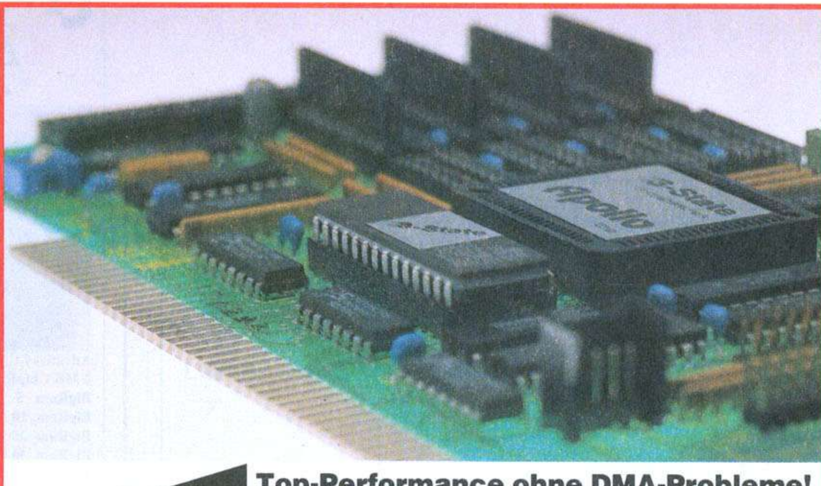
MegaMix 500 RAM-Box mit 2.0 MB **299,-**

MegaMix 2000 RAM-Karte mit 2.0 MB **249,-**

PREIS-HIT !

je weitere 2.0 MB **150,-**

APOLLO 500/2000



- 16 Bit-SCSI2-Controller
- 16 Bit-AT Bus/IDE-Controller
- 2/4/6/8 MB RAM-Expansion

Top-Performance ohne DMA-Probleme!

– Höchste Geschwindigkeit durch neuen 3-State Custom-Chip: Übertragungsrate bis zu 1.6MB/sec mit 68000-CPU
– In Verbindung mit Turbokarten setzt Apollo neue Maßstäbe! Statt langsamer 16Bit-DMA überträgt die CPU mit vollen 32Bit und erreicht mit Apollo Übertragungsraten von 2.5 MB/sec und mehr.

***NEU!* Jetzt mit Apollo-Software Version 2.0**

Volle Wechselplatten-Unterstützung mit Auto-Diskchange
Write-Cache und Read-Prefetch-Cache für superschnelle File-Operationen
Chamäleon-Support (Atari-ST-Emulator)
SCSI-Direct & AT-Direct

Apollo 2000 ohne RAM/HD	299,-	mit Harddisk	42 MB	699,-
		mit Quantum	85 MB	799,-
		mit Quantum	127 MB	899,-
RAM-Erweiterung um 2 MB	150,-	mit Quantum	170 MB	1049,-
Apollo 500 ohne RAM/HD	349,-	mit Harddisk	42 MB	749,-
		mit Quantum	85 MB	849,-
		mit Quantum	127 MB	949,-
RAM-Erweiterung um 2 MB	150,-	mit Quantum	170 MB	1099,-



AT-APOLLO 500/2000

16-Bit AT-Bus-Controller für A500 oder A2000

AT-Apollo 2000			AT-Apollo 500		
ohne HD		**199,-**	**ohne HD**		**249,-**
mit Harddisk	42 MB	599,-	mit Harddisk	42 MB	649,-
mit Quantum	85 MB	699,-	mit Quantum	85 MB	749,-
mit Quantum	127 MB	799,-	mit Quantum	127 MB	849,-
mit Quantum	170 MB	949,-	mit Quantum	170 MB	999,-

RAM-Option 2-8 MB für AT-Apollo 500, mit 2.0 MB **299,-**

Händlerdistribution Inland/Ausland

Sie sind Computerfachhändler mit einem Versand- oder Ladengeschäft und wollen auch AMIGA- und 3-State-Fachhändler werden. Dann wenden Sie sich mit Gewerbenachweis an unseren Distributor, Sie erhalten umgehend weitere Informationen.

Händlerdistribution:
Colossus Computer GmbH
Daimlerstr. 6b
W-4650 Gelsenkirchen 2
Fax: 02 09/77 92 36

Grafikstandards

EGS – was ist das?

Die Zahl der Grafikkarten für den Amiga steigt. Das veranlaßte einige Entwickler, eine uniforme Programmierschnittstelle zu schaffen. Ein Hintergedanke war es, die auf anderen Computern herrschende Verwirrung, durch viele Grafikkarten und Programme mit verschiedenen Softwaretreibern, zu vermeiden.

von Stefan Herr, Thomas Pfrengle und Ulrich Sigmund

Das Ergebnis ist EGS: »Extended Graphics Standard« für den Amiga. Die Entwickler versuchten, eine für das Amiga-System passende Lösung zu erarbeiten. Deshalb besteht EGS aus einer Sammlung von Bibliotheken (Shared Libraries), die auf Amiga-spezifische Weise benutzt werden. Weiterhin wurde versucht, sich an den bisher im Amiga-System existierenden Datenstrukturen und Funktionen zu orientieren, damit der bisher nur mit »Intuition« und »Graphics« beschäftigte Programmierer vieles aus seiner vertrauten Umgebung wiederfindet.

Die EGS-Bedieneroberfläche läßt sich mit Hilfe von VoreinstellungspProgrammen (»EGS Preferences«) variieren. Sie ermöglichen es sogar, die EGS-Oberfläche nach den Commodore-Richtlinien (»Intuition Interface Style Guide«) zu gestalten. Unter EGS laufende Programme sind vorbereitet, die vom Benutzer definierten Vorgaben zu verwenden. Hiermit ergibt sich die größtmögliche Einheit der Bedieneroberfläche.

EGS-Features

EGS hat eine frei konfigurierbare Oberfläche; Fenster lassen sich aus dem sichtbaren Bildschirmbereich herausschieben; virtuelle und reale 24-Bit True-Color-Grafik; Fenster, Gadgets, Menüs usw. sind Font-Sensitiv; Pop-Up-Menüs; am Fenster montierte und bewegliche Menüs; Menüs lassen sich von verschiedenen Fenstern teilen; gesteigerte Performance von EGS durch optimiertes Layering; Fenster kann man mit Inhalt verschieben; Verfügbar für die Amiga-Chipsätze »Standard« und »ECS« (»AA« in Kürze); folgende Grafikkarten unterstützen EGS: »VISIONA«, »EGS 110«, »IV-24«, »RAINBOW II«, »RAINBOW III«, »RAINBOW III entry«, »DOMINO« (über Zweitanbieter), »COLORMASTER« in Kürze über Zweitanbieter.



Die EGS-Oberfläche: True-Color-Darstellung bedeutet, daß immer mit 24 Bit bzw. 16,8 Millionen Farben gearbeitet wird – unter EGS kein Problem

Virtuelle 24-Bit-Umgebung

EGS-typisch ist, daß intern stets mit 24-Bit Farbtiefe (»True Color«) gearbeitet wird. Besitzt jedoch die momentan verwendete Grafikkarte bzw. der aktuelle Bildschirmmodus keine True-Color-Darstellung, werden die intern verwendeten 24-Bit-Farben in die darstellbaren umgerechnet. Hierzu verwendet EGS die Dithering-Technik sowie ein intelligentes Farbauswahlverfahren.

Der Vorteil: Die verfügbaren Farben eines Bildschirmmodus werden voll genutzt. Weiterhin vereinfacht dieses Prinzip die Entwicklung von Grafiksoftware, da ein Programm immer nur auf eine Farbtiefe (und zwar 24 Bit) eingehen muß.

Architektur von EGS

EGS ist eine Sammlung von Laufzeitbibliotheken, die wie Standard-Exec-Bibliotheken zu benutzen sind. Sie dienen dem hardwareunabhängigen Gebrauch von Grafikkarten auf dem Amiga. Sie stellen hierzu eine Vielzahl von Funktionen bereit, die Manipulationen des Bildschirminhalts einer Grafikkarte auf mannigfaltige Weise ermöglichen. Diese Manipulationen können auf vier verschiedenen Ebenen stattfinden:

❒ Auf der höchsten Ebene 4 finden man Bibliotheken, die den Entwurf von Bedieneroberflächen erleichtern. Sie schaffen via einfacher Beschreibungssprache Kommunikationsselemente, z.B. Gadgets und Requester.

❒ Auf der nächsttieferen Ebene gibt es Bibliotheken, die Funktionen zur Verwaltung der Bedieneroberflächen bereitstellen. Hierzu gehören die in Ebene 3 dargestellten Bibliotheken (s. Bild). Sie enthalten Funktionen, die eine ähnliche Benutzerschnittstelle wie Commodores Intuition implementieren, z.B. Screens oder Windows.

❒ Auf der darauf folgenden Ebene stehen Bibliotheken mit Funktionen zur direkten Manipulation von Bitmaps und ihrer Überlagerung (Layers) zur Verfügung. Dies entspricht Ebene 2. Die hier vorhandenen Bibliotheken haben ihr Pendant in Commodores Graphics- und Layers-System.

❒ Auf der untersten EGS-Ebene (Ebene 1) befinden sich die für die Abstrahierung der Grafikhardware zuständigen Bibliotheken. Sie stellen Low-Level-Zeichenfunktionen sowie Funktionen zur prinzipiellen Verwaltung der Grafikkarte und ihrer Darstellungsmodi bereit (EGS.library). Weiterhin existieren hier noch Funktionen zur Verschiebung und Konvertierung von Grafikspeicherausschnitten (EGSBlit.library).

Alle oben erwähnten Ebenen basieren auf der Grafikkartenhardware sowie dem Amiga-Betriebssystem an sich (hauptsächlich der exec.library). Das entspricht Ebene 0.

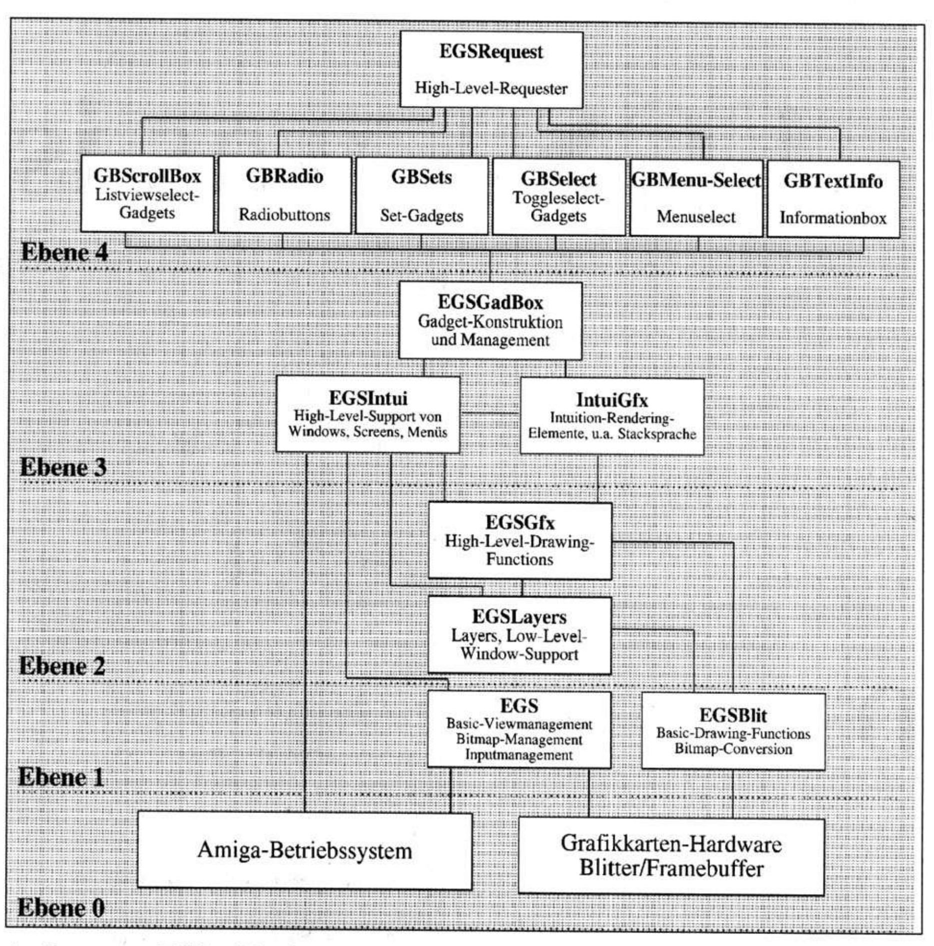
Je nachdem, welche Komplexität bzw. Anforderungen (Geschwindigkeit, Flexibilität) ein zu entwickelndes Programm fordert, entscheidet sich der Programmierer für eine der vier EGS-Ebenen. Am komfortabelsten und somit wohl am häufigsten benutzt werden die oberen beiden Ebenen 3 und 4. Für einfache Grafikausgaben, die auf ein Fenstersystem verzichten können, verwendet man Ebene 2. Die untere Ebene ist dafür gedacht, besonders schnell mit der Grafikhardware zu arbeiten (allerdings immer noch hardwareunabhängig).

Kompatibilität und Portierbarkeit

Für etliche Grafikkarten und auch die Amiga-Grafikchipsätze (Standard, ECS) existiert eine EGS-Anpassung. Die fürs AA-Chipset folgt in Kürze. Hierfür müssen lediglich die »egs.library« und »egsblit.library« für die jeweilige Grafikhardware angepaßt werden. Programmen bleibt dabei völlig verborgen, welche Grafikkarte verwendet wird. Ein EGS unterstützendes Programm läuft ohne Änderung auf allen von EGS unterstützten Grafikplattformen.

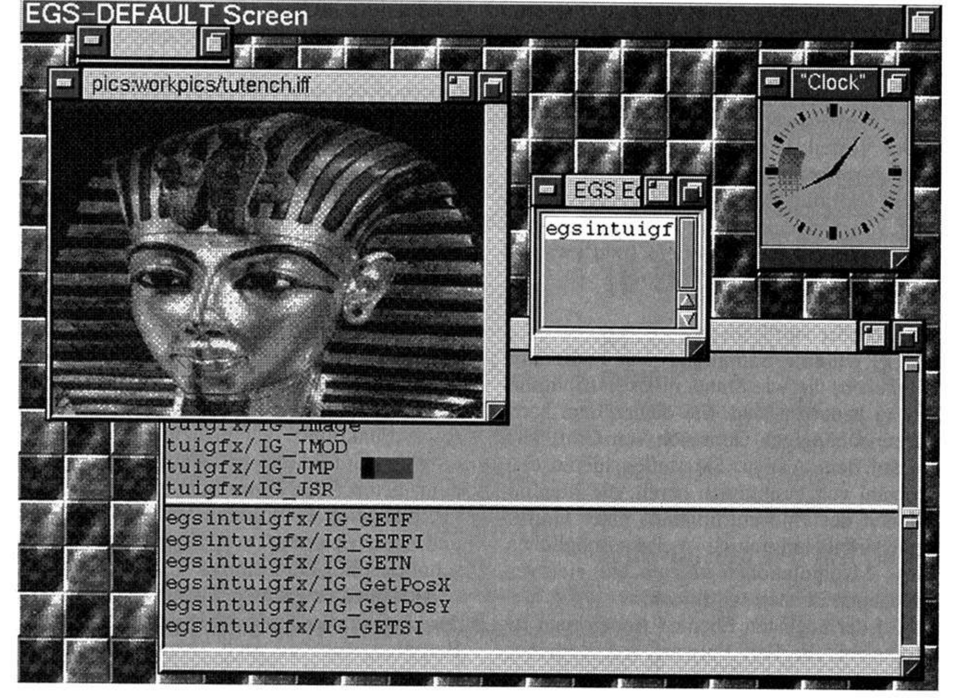
Grundsätzlich ist jedes für Intuition oder Graphics geschriebenes Programm aufs EGS-System portierbar. EGS stellt gerade hierfür ähnliche Funktionen und Datenstrukturen wie das Amiga-System bereit. Einige Dinge sind jedoch zu beachten:

❒ Viele Datenstrukturen wurden erweitert und in vielen Fällen erheblich in der Funktionalität ergänzt. Dies folgt daraus, daß EGS auf 24 Bit (True Color) beruht, was naturgemäß einen größeren Leistungsumfang bedingt. Es kann daher nicht überall eine 1:1-Übereinstimmung mit den Amiga-Datenstrukturen erwartet werden. Ähnliches gilt auch für die Bibliotheksfunktionen und ihren Aufruf.



Aufbau von EGS: Die fünf verschiedenen Ebenen bieten für jede Softwareanpassung die gewünschten Fähigkeiten und Eigenschaften

❒ Es bestehen gegenüber dem Amiga-System Unterschiede in der Bezeichnung der Datenstrukturen und ihrer Elemente sowie der Bezeichnung der Funktionen und ihrer Argumente: So ist eine bessere Unterscheidung der EGS-Bezeichner bei gleichzeitiger Verwendung der dem Amiga-System zugehörigen möglich.



Vorteile von EGS

Die Portierung eines Programms auf EGS bietet einige Vorteile, u.a. die völlige Unabhängigkeit der Programme von der Grafikhardware sowie der Unabhängigkeit von der Grafikdarstellung, Auflösung und Farbtiefe.

Vertriebsweise

Das EGS-System wird von den Grafikkartenherstellern lizenziert und vertrieben. Somit muß ein Programmierer für EGS keine Gebühren für Nutzungsrechte entrichten. Auch die Käufer der Grafikkarten profitieren von dieser Verfahrensweise, da sich die von ihnen gekaufte EGS-Software auf allen Karten gleich verhält und auch gleich aussieht.

Haben Sie Interesse, Programme unter der EGS-Oberfläche zu entwickeln, wenden Sie sich bitte an:

Viona Development
Dreiherrenstein 6a
6200 Wiesbaden-Auringen

Sie erhalten dort weitere Informationen und Unterstützung. Softwareentwickler erhalten die Amiga-Implementation unentgeltlich.

rz

Portable Applikationen

Amiga vs. Windows

Seitdem es Windows 3.0 gibt, wollen auch die PC-User ihre Maschinen nur noch mit der Maus dirigieren. 13 Millionen Pakete weltweit wurden schon verkauft und monatlich kommt eine Million hinzu. Da sollte es sich lohnen, Amiga-Programme umzuschreiben. Wir sagen Ihnen, was Sie dabei erwartet.

von Peter Wollschlaeger

Wir wollen ja nicht lästern: Daß die grafische Bedienoberfläche des Amiga der schlichten DOS-Shell haushoch überlegen ist, wissen wir schließlich selbst seit Jahren am besten. Fragt sich nur, warum die PC-Gemeinde erst jetzt umschwenkt? Die Antwort: Vorher gab es nur eine Pseudografik im Textmodus, und die überzeugte niemanden. Erst Windows 3.0 brachte dem PC das Amiga-Feeling, und seitdem – oder deshalb – ist Windows das erfolgreichste Softwarepaket aller Zeiten.

Wie beim Amiga basiert Windows auf dem DOS und wird sinngemäß wie die Workbench gestartet, nachdem das System in der Startup-sequence (»AUTOEXEC.BAT« und »CONFIG.SYS«) konfiguriert wurde. Von Haus aus meldet es sich mit dem Programm-Manager (Bild 1). Ein anderes Startprogramm einzutragen, bedeutet lediglich, eine Zeile in der Textdatei »WIN.INI« anzupassen. Wenn Sie sich also eine bessere Oberfläche produzieren wollen – noproblem: Doch in diesem Markt gibt's schon einige Wettbewerber.

Windows ist ein »non preemptive«-Multitasking-System, sieht man einmal davon ab,

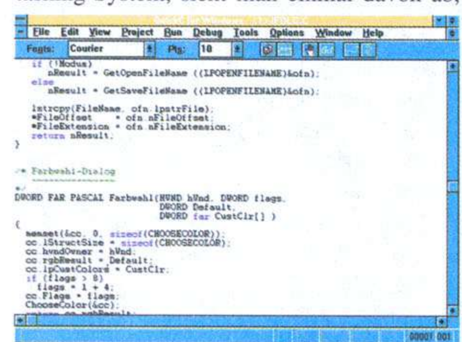


Bild 2: Gute Entwicklungssysteme für Windows sind selbst Windows-Applikationen. Abgebildet ist der C-Compiler »Quick C/Windows« (Microsoft).

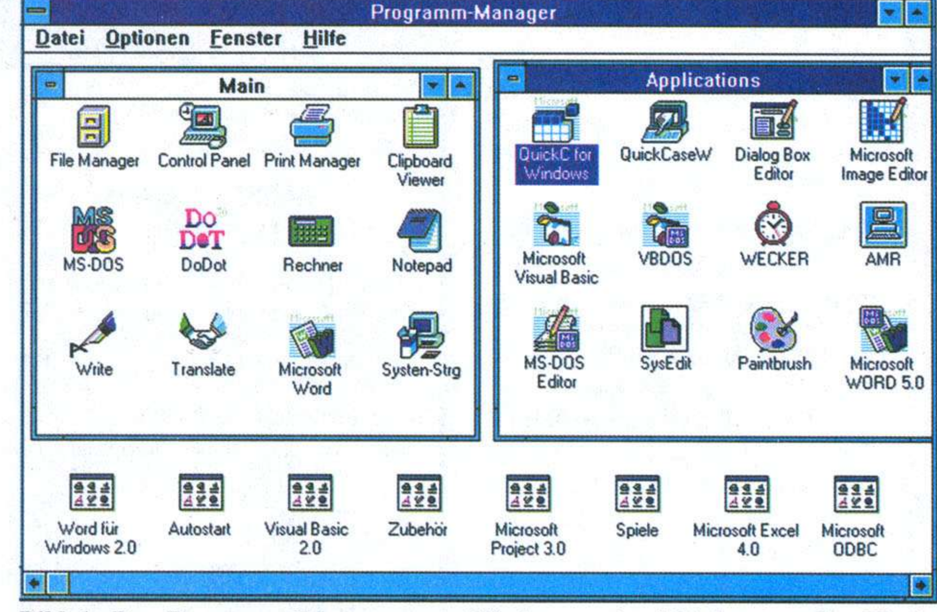


Bild 1: Der Programm-Manager von Windows entspricht in etwa der Workbench des Amiga. Die tolle Farbenvielfalt gibt's unter OS 2.0/3.0 jetzt auch.

daß auf 386ern mehrere DOS-Tasks (keine Windows-Tasks) wirklich parallel laufen können. Im Normalfall hat von mehreren offenen Programmen nur eines den Fokus, verfügt also über die Tastatur, die Maus und den Bildschirm und kann die Kontrolle für alle Ewigkeiten behalten. Einzige Ausnahme: Das Programm fragt Windows, ob der User die Tastatur oder die Maus betätigt hat. In diesem Fall geht die Kontrolle an Windows zurück, das dann einem anderen Task die Systemressourcen zuteilen kann.

Dieses kooperative Multitasking-System setzt voraus, daß sich die Anwendungsprogrammierer an einige Spielregeln halten; beim Amiga ist das ähnlich. So sollte man in zeitintensiven Programmteilen mit einem Funktionsaufruf periodisch die Kontrolle an Windows übergeben oder im Minimum den User durch die Änderung des Mauszeigers in ein Sanduhr-Symbol informieren.

Noch zwei Unterschiede: Windows kennt zwar zahllose Bildschirmauflösungen und Farbtiefen von monochrom bis über 16 Millionen Farben – ändert man sie allerdings, ist Windows neu zu starten. Während des Betriebs gibt's nur einen Screen. Der zweite Unterschied ist hardwarebedingt. Die Soundfähigkeiten des PC sind praktisch gleich Null. Ohne Soundkarte, und die hat kaum jemand, können Sie das Thema vergessen.

Doch nun zur Praxis. Listing 1 zeigt ein simples Intuition-Programm, das ein Fenster öffnet und »Hallo Welt!« hineinschreibt. Dazu wird eine Window-Struktur mit Daten gefüllt. Diese Pflichtkür werden Sie auch im Windows-Listing antreffen. Der Unterschied beginnt beim Öffnen der Libraries. Dazu gibt es zwar mit der »DLL« (Dynamic Link Library) ein Pendant in Windows, doch die Übereinstimmungen sind gering. DLLs werden von den Anwendungsprogrammierern geschrieben und von Windows verwaltet. Die Folge: Kein Windows-Programm benötigt eine DLL, und ein Programm muß eine DLL auch nicht schließen. Die Einstimmigkeit besteht lediglich darin, daß auch eine DLL nur einmal geladen wird und dann allen Tasks zur Verfügung steht.

Nachdem das Window geöffnet ist, wird ein Rastport beschafft. Dessen Windows-Pendant heißt »DC« (Device-Context), und wie hier, verlangen auch alle Windows-Grafikfunktionen den DC als Argument. Der Unterschied: Die Standardtextausgabe »TextOut()« will auch gleich die Cursor-Koordinaten sehen. Listing 1 wartet dann mit »Wait()« auf eine Close-Message und endet.

Die gleiche Aufgabe in Windows löst Listing 2. Das Programm ist zwar Ihnen zuliebe etwas ausführlicher kommentiert, doch eines stimmt schon: Windows-Programme sind länger. Das erste Problem: Sie schreiben relativ viel Code, den Sie selbst nie aufrufen. In der Praxis stört das nicht, denn diese Blöcke lassen sich als Include-Dateien

oder als DLL einbeziehen oder mit einem CASE-Tool automatisch erzeugen, doch das Prinzip möchte erst einmal verstanden sein. Die Funktionen, die Sie nie selbst aufrufen, nennt man Callback-Funktionen. Sie sind sehr einfach am Schlüsselwort »PASCAL« oder »CALLBACK« zu erkennen. Das bedeutet lediglich, daß die Parameterübergabe nicht nach der C-, sondern nach Pascalkonvention erfolgt (von links nach rechts). Diese Vorgehensweise hat den Vorteil, daß sie schneller ist und das aufrufende Programm den Stack nicht korrigieren muß. Der Nachteil: eine variable Anzahl von Funktionsargumenten ist in Pascal nicht möglich.

Die erste Callback-Funktion »WinMain()« ist logisch: Es ist der Entry-Point des Programms. Die Funktion ruft Windows beim Programmstart auf. Der Name WinMain ist obligatorisch, alle anderen Funktionsnamen dürfen Sie selbst vergeben. Schon im Funktionskopf tauchen einige Begriffe auf, die Sie kennen müssen. Da ist zuerst »Handle«. Nahezu alle Objekte in Windows, ob Fenster, Menüs, Fonts und sogar Speicherblöcke, werden über Handles angesprochen. Ein Handle ist nichts weiter als eine Kennzahl vom Typ Wort, die Windows verteilt und verwaltet.

Klassengesellschaft

Der nächste Unterschied sind die Instanzen. Ein Windows-Programm läßt sich mehrmals starten. Möchten Sie beispielsweise zwei verschiedene Texte gleichzeitig bearbeiten, öffnen Sie den Editor zweimal, und schon lassen sich Textausschnitte über die Zwischenablage transportieren. Doch tatsächlich werden nur weitere Datenbereiche angelegt, der Code befindet sich nur einmal im Speicher. Das Problem dabei: Diese Instanzen müssen dieselbe Windows-Klasse nutzen wie das erste Programm, womit wir erst einmal diesen Punkt aufklären müssen. Sie können nicht direkt ein Window anlegen, sondern müssen zuerst eine Klasse schaffen, man sagt auch, die Klasse muß (von Windows) registriert werden. Von einer Klasse lassen sich dann diverse Fenster(-Objekte) ableiten. Die Fenster können unterschiedlich aussehen, doch bestimmte Eigenschaften erben alle Fenster von ihrer Klasse.

Zurück zu den Instanzen: Wenn Windows WinMain beim Programmstart aufruft, kann das sein erster Versuch sein, es kann aber damit auch die zehnte Instanz starten wollen. Deshalb übergibt Windows in »hInstance« das Handle der neuen Instanz und in »hPrevInstance« das Handle des Vorgängers. Wenn letzteres Handle Null ist, gibt es noch keinen Vorgänger und es ist der erste Start. Nur in diesem Fall darf eine neue Klasse registriert werden, was hier durch den Aufruf der Funktion »nCwRegisterClasses()« geschieht.

Die Bits »CS_HREDRAW« und »CS_-VREDRAW« bestimmen, daß das Fenster aktualisiert werden soll, wenn der Anwender die horizontalen oder die vertikalen Rollbalken einsetzt. Hier steckt also die Refresh-Methode, die man unter Intuition beim Window-Anlegen definiert. Hier definiert man auch den Default-Mauszeiger, legt die Hintergrundfarbe fest und definiert das Default-Menü für alle Instanzen.

Zurück zur WinMain-Funktion: Egal, ob die Klasse neu angelegt wurde oder ob es Sie schon gibt, auf jeden Fall wird jetzt ein erstes oder ein weiteres Fenster benötigt. Das macht »CreateWindow()«. Auffallend ist die Ähnlichkeit mit dem »OpenWindow()«-Aufruf von Intuition, nur daß hier die Steuer-Bits für die Gadgets anders bezeichnet sind. Windows gibt für das neue Fenster ein Handle zurück oder Null, wenn etwas beim Anlegen schiefging. In letzterem Fall wird mit »MessageBox()« ein Dialog auf den Schirm gebracht. Hier ist anzumerken, daß unter Windows Dialoge (Requester) wesentlich einfacher anzulegen und handzuhaben sind.

Nach dem »ShowWindow()«-Aufruf ist das Fenster auf dem Schirm und das Programm geht in die Message-Loop. Die WHILE-Schleife läuft und läuft, bis eine Quit-Message eintrifft. Messages werden mit

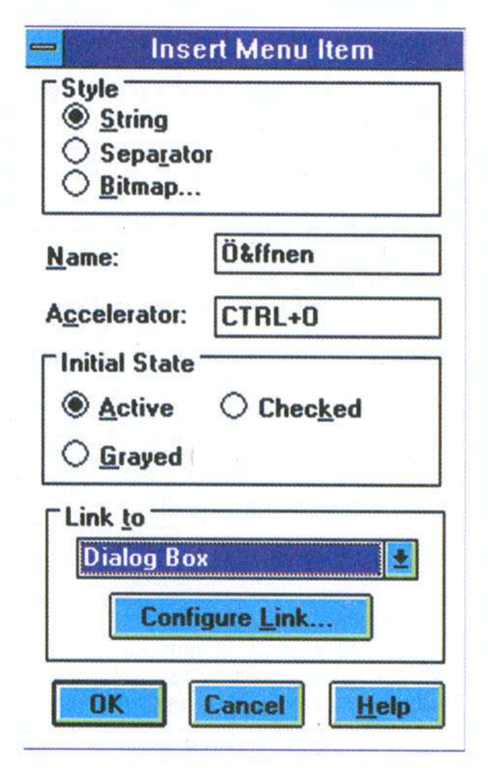


Bild 3: Die gesamte Oberfläche wird mit Ressource-Editoren erzeugt. Hier wird beispielsweise ein Menüeintrag mit einem solchen Editor definiert.

```
 1: #include <exec/types.h>
 2: #include <intuition/intuition.h>
 3:
 4: struct IntuitionBase *IntuitionBase;
 5: struct GfxBase *GfxBase;
 6:
 7: /*  Struktur für neues Window initialisieren:  */
 8: struct NewWindow NewWindow =
 9: {
10:     170,80,              /* linke obere Ecke    */
11:     300,100,             /* Breite u. Höhe      */
12:     -1,-1,               /* Farbe der Pens      */
13:     CLOSEWINDOW,         /* Meldung wenn        */
14:
15:     WINDOWCLOSE   |      /* Window-Gadgets      */
16:     WINDOWSIZING  |
17:     WINDOWDRAG    |
18:     WINDOWDEPTH   |
19:     SMART_REFRESH |      /* Refresh-Art          */
20:     ACTIVATE,            /* aktiv nach Open()    */
21:
22:     NULL,                /* keine User-Gadgets   */
23:     NULL,                /* keine User-CheckMark */
24:     "Mein Window",       /* Window-Titel         */
25:     0,                   /* Kein eigener Screen  */
26:     NULL,                /* keine SuperBitmap    */
27:     100,                 /* Mindestbreite        */
28:     30,                  /* Mindesthöhe          */
29:     640,                 /* Maximalbreite        */
30:     256,                 /* Maximalhöhe          */
31:     WBENCHSCREEN         /* Bildschirmtyp        */
32: };
33:
34: main()
35: {
36:     struct Window *Window;
37:     struct RastPort *rp;
38:
39:     /* Zwei Libraries öffnen, bei Fehler Exit: */
40:     IntuitionBase =
41:     (struct IntuitionBase *) OpenLibrary("intuition.library",0L);
42:     if (IntuitionBase == NULL)  exit(FALSE);
43:
44:     GfxBase =
45:     (struct GfxBase *) OpenLibrary("graphics.library",0L);
46:     if (GfxBase == NULL)
47:     {
48:        CloseLibrary(IntuitionBase);
49:        exit(FALSE);
50:     }
51:
52:     /* Window öffnen, bei Fehler Exit: */
53:     Window =
54:     (struct Window *) OpenWindow(&NewWindow);
55:
56:     if (Window == NULL)
57:     {
58:        CloseLibrary(GfxBase);
59:        CloseLibrary(IntuitionBase);
60:        exit(FALSE);
61:     }
62:
63:     /* Rastport beschaffen und Text ausgeben:      */
64:     rp = Window->RPort;                /* Der Rastport  */
65:     SetAPen(rp, 1L);                   /* Farbe weiß    */
66:     Move(rp, 100, 50);                 /* Cursor -> x,y */
67:     Text(rp, "Hallo Welt!",11L);  /* Text zeichnen */
68:
69:     /* Auf Message warten,
70:        kann hier hier nur Close sein: */
71:     Wait(1L << Window->UserPort->mp_SigBit);
72:
73:     CloseWindow(Window);          /* Alles schließen */
74:     CloseLibrary(GfxBase);
75:     CloseLibrary(IntuitionBase);
76:
77:     exit(TRUE);                   /*   und Ende      */
78: }                  © 1993 M&T
```

Listing 1: Ein simples Intuition-Programm, das ein Fenster öffnet und »Hallo Welt!« hineinschreibt

»DispatchMessage()« an die Funktion »WndProc()« gesendet.

Da fragt man sich zunächst, warum ausgerechnet dorthin? Die schlichte Antwort: Weil Sie beim Registrieren der Klasse den Zeiger auf diese Funktion in die Klassenstruktur eingetragen haben.

Also geht's jetzt bei »WndProc()« weiter. Die MSG-(Message-) Struktur an sich entspricht ziemlich genau der von Intuition inklusive Uhrzeit und Mauskoordinaten. »DispatchMessage()« löst diese Struktur jedoch auf und sendet nur noch den Typ und zwei Parameter (entspricht der Message-Klasse und dem Code in zwei Gruppen). Auch WndProc() ist eine Callback-Funktion, wird also nur von Windows aufgerufen, doch hier steht auch der Kern Ihres Programms. In diesem einfachen Beispiel sind die Handler für die beiden Messages »WM_PAINT« und »WM_CLOSE« direkt in den Switch-Selektor geschrieben worden – normalerweise wird man hier jedoch Funktionsaufrufe einsetzen.

Der wichtigste Fall ist »WM_PAINT«. Diese Message wird immer dann gesendet, wenn ein Fenster neu zu zeichnen ist, also zuerst nach dem Start, nach jedem Verschieben, nach jeder Größenänderung und in noch einigen Fällen. Was genau anliegt, finden wir in der »PAINTSTRUCT«-Struktur, z.B. das Rechteck, das zu aktualisieren ist. Die Struktur wird an »BeginPaint()« übergeben, womit weitere Änderungen gesperrt werden. Ist das Fenster neu gezeichnet, gibt man es mit »EndPaint()« wieder frei.

Als nächste Message wertet man »WM_CLOSE« aus. Man beachte den feinen Unterschied zu Intuition: »DestroyWindow()« löscht das Fenster der aktuellen Instanz. Anschließend prüft man, ob das die letzte (oder einzige) Instanz war. Nur in diesem Fall wird mit »PostQuitMessage(0)« die Ende-Nachricht in den Message-Queue gepackt.

Wie gesagt, »WndProc()« wird von »WinMain()« aufgerufen, also geben wir die Kontrolle dorthin zurück. Hier findet die Message-Loop die Null- oder Quit-Message vor

```
  1: #include <windows.h>
  2: #include <string.h>
  3:
  4: char szAppName[20];    // Klassen-Name für dieses Window
  5: HWND hInst;            // Instanz-Handle
  6: HWND hWndMain;         // Window-Handle
  7:
  8: // Prototypen (siehe unten)
  9: LONG FAR PASCAL WndProc(HWND, WORD, WORD, LONG);
 10: int nCwRegisterClasses(void);
 11: void CwUnRegisterClasses(void);
 12:
 13:
 14: int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
 15:                    LPSTR lpszCmdLine, int nCmdShow)
 16: {
 17:  /***************************************************
 18:   * Hier startet das Programm.                      *
 19:   * Diese Funktion wird nur von Windows aufgerufen! *
 20:   ***************************************************/
 21:
 22:  MSG        msg;  // Message-Struktur
 23:  int        nRc;  // Return-Wert von RegisterClasses
 24:
 25:  strcpy(szAppName, "Demo");
 26:  hInst = hInstance;
 27:  if(!hPrevInstance)
 28:    {
 29:     // Wenn es die erste Instanz ist, Klasse registrieren
 30:     if ((nRc = nCwRegisterClasses()) == -1)
 31:       {
 32:        // Wenn Fehler beim Registrieren
 33:        MessageBox(NULL, "Konnte Klase nicht registrieren",
 34:                   "Sorry", MB_ICONEXCLAMATION);
 35:        return nRc;
 36:       }
 37:    }
 38:
 39:  // Klasse gibt es, also Window anlegen:
 40:  hWndMain = CreateWindow(
 41:             szAppName,          // Name der Applikation
 42:             "Text des Titels", // Das sog. Caption
 43:             WS_CAPTION       | // Hat Titelleiste und
 44:             WS_SYSMENU       | // System-Menü und
 45:             WS_MINIMIZEBOX   | // Minimum-Box und
 46:             WS_MAXIMIZEBOX   | // Maximum-Box und
 47:             WS_THICKFRAME    | // Malt nicht in Child-Windows
 48:             WS_OVERLAPPED,     // Kann überlappen
 49:             CW_USEDEFAULT, 0,  // Default für links oben
 50:             CW_USEDEFAULT, 0,  // Default für rechts unten
 51:             NULL,              // Ist keine MDI-Applikation
 52:             NULL,              // Default ist Klassen-Menü
 53:             hInst,             // Instanz dieses Windows
 54:             NULL);             // Kein Struct für WM_CREATE
 55:
 56:
 57:  if(hWndMain == NULL)
 58:    {
 59:     MessageBox(NULL, "Konnte Window nicht anlegen",
 60:                "Sorry", MB_ICONEXCLAMATION);
 61:     return 1;
 62:    }
 63:
 64:  ShowWindow(hWndMain, nCmdShow);     // Window anzeigen
 65:
 66:  while(GetMessage(&msg, NULL, 0, 0)) // Endlos
 67:    {
 68:     TranslateMessage(&msg);              // Keys übersetzen
 69:     DispatchMessage(&msg);               // Message an WndProc
 70:
 71:    }                                 // bis WM_Quit-Message
 72:
 73:  // Rückzugsgefechte und Ende
 74:  CwUnRegisterClasses();
 75:  return msg.wParam;
 76: } //  End of WinMain
 77:
 78:
 79: /*********************************************
 80:  *                                           *
 81:  * WndProc (Main Window Procedure)           *
 82:  *                                           *
 83:  * Hierher sendet Windows Messages über alle *
 84:  * Events, egal ob das User-Aktionen sind oder *
 85:  * Windows-Messages.                         *
 86:  *********************************************/
 87:
 88: LONG FAR PASCAL WndProc(HWND hWnd, WORD Message,
 89:                         WORD wParam, LONG lParam)
 90: {
 91:  HDC         hDC;    // Handle für Display-Device-Context
 92:  PAINTSTRUCT ps;     // Hält PAINT-Informationen
 93:  int         nRc=0;  // Return-Code
 94:
 95:  switch (Message)
 96:    {
 97:     case WM_PAINT: // Neuzeichnen evtl. nötig
 98:          memset(&ps, 0x00, sizeof(PAINTSTRUCT));
 99:          hDC = BeginPaint(hWnd, &ps); // Melde Start
100:          SetBkMode(hDC, TRANSPARENT); // Hintergrund löschen
101:          TextOut(hDC, 20, 20, "Hallo Welt!", 11);
102:          EndPaint(hWnd, &ps); // Melde, daß Painting fertig
103:          break;
104:
105:     case WM_CLOSE: // User schließt Window
106:          DestroyWindow(hWnd);   // Window abbauen.
107:          if (hWnd == hWndMain)  // Wenn es das letzte ist,
108:            PostQuitMessage(0);  // Ende melden
109:         break;
110:
111:     default:
112:          // Für alle Messages, die keine Service-Routine haben,
113:          // Messages an Windows für Default-Prozess zurück:
114:          return DefWindowProc(hWnd, Message, wParam, lParam);
115:    }
116:  return 0L;
117: }     // End of WndProc
118:
119: /***************************************************
120:  *                                                 *
121:  * Generelle Funktion.                             *
122:  *                                                 *
123:  * Registriert alle Klassen aller Windows,         *
124:  * die zur Applikation gehören.                    *
125:  * Return: 0, wenn erfolgreich, sonst Error-Code *
126:  *                                                 *
127:  ***************************************************/
128:
129: int nCwRegisterClasses(void)
130: {
131:  WNDCLASS   wndclass; // Struct der Windowsklasse
132:  memset(&wndclass, 0x00, sizeof(WNDCLASS));
133:
134:
135:  // Stuktur laden:
136:  wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_BYTEALIGNWINDOW;
137:  wndclass.lpfnWndProc = WndProc;
138:  // Kein Extra-Storage für Klassen- und Windows-Objekte:
139:  wndclass.cbClsExtra = 0;
140:  wndclass.cbWndExtra = 0;
141:  wndclass.hInstance = hInst;
142:  wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
143:  wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
144:  // Brush für Hintergrund-Löschen anlegen:
145:  wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
146:  wndclass.lpszMenuName = szAppName;  // Menü-Name = App.-Name
147:  wndclass.lpszClassName = szAppName; // Klassen-Name = App.-Name
148:
149:  return RegisterClass(&wndclass); // Registrieren
150: } // End of nCwRegisterClasses
151:
152: /*********************************************
153:  *  CwUnRegisterClasses-Funktion             *
154:  *                                           *
155:  *  Löscht alle Refrenzen zu den Resourcen  *
156:  *  und alle Handles und gibt Speicher frei.*
157:  *********************************************/
158:
159: void CwUnRegisterClasses(void)
160: {
161:  WNDCLASS   wndclass;
162:  memset(&wndclass, 0x00, sizeof(WNDCLASS));
163:
164:  UnregisterClass(szAppName, hInst);
165: }
```


Listing 2: Das Windows-Pendant zu Listing 1 ist nicht nur wegen der ausführlicheren Kommentare länger

und endet. Das Programm setzt bei »cwUnRegisterClasses()« fort – wir finden die Funktion am Listingende. Hier wird nur die Klasse und damit der gesamte Speicher der Applikation freigegeben, in der Praxis finden hier noch ein paar weitere »Rückzugsgefechte« statt.

Sehr gute Entwicklungssysteme

Nach den ersten Grundlagen nun etwas zur Umgebung, in der Windows-Programme entwickelt werden. Sie können in »Visual Basic« (herrlich einfach), in C, C++ oder in Pascal arbeiten. In allen Fällen ist es dem Programm nicht anzusehen, welcher Compiler es erzeugt hat, da alle Entwicklungssysteme auf dasselbe Windows-API zugreifen. Dieses »API« ist mit über 600 Funktionen deutlich leistungsfähiger als das von Intuition, und wird noch durch einige DLLs, z.B. für alle Standarddialoge, ergänzt. Hier liegt allerdings auch die Hürde, denn die 600 Funktionen plus Hunderte von Messages muß man erst einmal kennen.

Am schnellsten kommt man mit Windows in der Programmiersprache C klar, denn das ist (wie bei Intuition) seine Muttersprache. Die gesamte Dokumentation des »SDK« (Software Development Kit) von Microsoft – es sind ja nur 5000 Seiten – basiert auf C. Sofern Sie nicht schon sehr fit in C++ sind, sollten Sie mit C starten, und zwar am einfachsten mit »Quick C« für Windows. Bild 2 demonstriert, daß dessen Entwicklungsumgebung selbst eine Windows-Applikation ist, weshalb Sie, ohne die Umgebung verlassen zu müssen, Programme sofort austesten und debuggen können. Zu »QC/Win« gehört auch ein CASE-Tool, mit dem sich die gesamte Bedieneroberfläche eines Programms, also Menüs und Dialoge, mit der Maus zusammenstellen lassen. Bild 3 zeigt, wie auf diese Art Menüs entstehen und Bild 4 ist ein Schnappschuß des Dialogeditors. Ein Imageeditor für Bildchen oder eigene Mauszeiger gehört noch dazu.

Code und Ressourcen strikt trennen

Das Ergebnis des CASE-Tools ist C-Quelltext plus einem Make-File, der die zusätzlich erzeugten Quelltexte dem Compiler anbietet, womit wir beim nächsten Unterschied wären: In Windows gibt es eine ganz klare Trennung zwischen Code und Ressourcen – jedenfalls sollte es so sein. In Listing 2 sind sämtliche Stringliterale, also die Texte für Message-Boxen oder das »"Hallo Welt"«, bereits ein grober Verstoß gegen die Spielregeln. Normalerweise programmiert man nicht so, doch das Listing sollte im ersten Ansatz nicht allzu kompliziert sein. Ressourcen sind alle Texte, die das Programm ausgibt, die Menüs, alle Dialoge, der Cursor bzw. der Mauszeiger, jeder Zeichensatz, jedes Bild: Kurz gesagt alles, was man sieht, mit Ausnahme der Fenster.

Die Ressourcen erzeugt man mit getrennten Tools, übersetzt sie mit einem Ressource-Compiler und bindet sie schließlich mit dem Programmcode zu einer ausführbaren Datei (».EXE«). Das ist auch der Grund, weshalb man ein US-Programm eindeutschen kann, ohne den Quelltext besitzen zu müssen. Spezielle Ressourceeditoren ziehen den Ressourcenteil aus der Exe-Datei heraus, wandeln ihn in Bilder oder Klartext um, erlauben das Editieren, Kompilieren und Linken. Zu empfehlen ist der »Resource Workshop« von Borland.

Auch die grafischen Tools erzeugen im Falle von Menüs, Strings und Dialogen Textdateien. Früher gab es die Editoren nicht, da tippte jeder die Text- und Header-Dateien selbst. Heutzutage editiert man nur bei kleineren Änderungen, z.B. einem neuen Menüpunkt, die Textdateien direkt.

Um den Unterschied im Umgang mit Menüs und Messages zu verdeutlichen, zeigt Listing 3 die Message-Loop und den Menü-Handler von Intuition und Listing 4 das Windows-Pendant. Letzteres ist jetzt aber nur länger, um bei der Gelegenheit noch eine Differenz zu nennen.

Zuerst sollte auffallen, daß man sich unter Intuition via »MENUPICK«, »MENUNUM«, »ITEMNUM« usw. zu dem einzelnen Menüpunkt vorarbeiten muß, während in Windows jeder Menüpunkt direkt angesprochen wird. Die Hierarchie verwaltet Windows selbst, was z.B. zur Folge hat, daß alle Items eines Menüs »disabled« sind, blendet man den Titel aus.

In Listing 4 – einem Auszug aus einem Texteditor – geht's nun los. Nachdem ein Menü aufgebaut oder eingeschaltet ist, sendet Windows eine »WM_INITMENU«-Message. Hier wird geprüft, ob Text in der Zwischenablage ist und wenn ja, der Menüpunkt »Einfügen« (Paste) aktiviert, andernfalls wird er grau gezeichnet (MF_GRAYED). Übrigens stammen diese und andere Konstanten sowie alle Prototypen aus der rund 140 KByte großen Headerdatei »windows.h«. Die Punkte »Neu«, »Öffnen« und »Sichern« schreibt man auf dem Amiga sinngemäß, nur daß man hier die Standarddialoge aus »CmnDlg.DLL« einsetzen sollte. Diese Dialoge folgen der alten Regel »Struktur laden, Funktion aufrufen«. Recht kompliziert hingegen ist das Drucken. Da es lange dauern kann, bis die Grafik einer Seite aufgebaut ist (auch Text ist nur Grafik), ist für die ungeduldigen Benutzer zunächst ein Abbruchdialog (-Requester) zu laden, einzubinden und bei Windows anzumelden. Dann muß man sich die Auflösung einer Druckerseite beschaffen, für jede Zeile die maximale Höhe ermitteln und so zu einer Zeilenzahl pro Seite kommen. Daß man noch einen kompatiblen DC anlegen muß, um gemäß der Duckerauflösung das Bild im Speicher aufbauen zu können, sich bei Bildern um eine kompatible Bitmap kümmern und sich auch noch Speicher beschaffen muß, sei nur am Rande

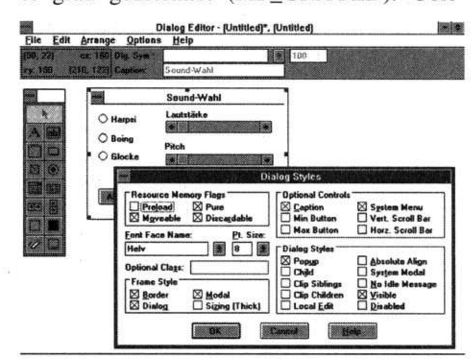


Bild 4: Dialogboxen sind sehr viel einfacher anzulegen und zu handhaben als die Requester von Intuition älterer Betriebssysteme. OS 2.0 hat aufgeholt.

```
 1: for(;;)  /* Message-Loop */
 2:    {
 3:       Wait(1L << Window->UserPort->
                              mp_SigBit);
 4:
 5:       while( msg = GetMsg(Window->UserPort)
)
 6:       {
 7:          class = msg->Class;
 8:          code  = msg->Code;
 9:          ReplyMsg( msg );
10:
11:          switch( class)
12:          {
13:             case CLOSEWINDOW:
14:                  close_all();
15:                  break;
16:
17:             case MENUPICK:
18:                  if (MENUNUM(code) !=
                              MENUNULL)
19:                         do_menu();
20:                       break;
21:          }
22:       }
23:    }
24:
25:
26: void do_menu()
27: {
28:    switch(MENUNUM(code)) /* Switch Titel */
29:      {
30:        case  IDM_DATEI:
31:              switch(ITEMNUM(code))
32:              {
33:                 case IDM_D_NEU:
34:                      Do_Neu();
35:                      break;
36:                 case IDM_D_OEFFNEN:
37:                      Do_Oeffnen();
38:                      break;
39:              }
40:              break;
41:
42:        case  IDM_BEARBEITEN:
43:              switch(ITEMNUM(code))
44:              {
45:                 case  IDM_B_UNDO:
46:                       do_Undo();
47:                       break;
48:                 case  IDM_B_COPY:
49:                       do_Copy();
50:                       break;
51:              }
52:              break;
53:      }
54: }

          © 1993 M&T
```

Listing 3: Die Message-Loop und der Menü-Handler sehen unter Intuition

erwähnt. Der nächste Punkt, nämlich die Druckerinstallation, verdeutlicht noch einmal, was hier alles zu berücksichtigen ist.

Das Problem: Der Druckerinstallations- oder Einrichten-Dialog (Auflösungswahl, Rändereinstellung, Anzahl Kopien u.ä.) gehört nicht zum Windows-API, sondern wird vom Druckerhersteller als Teil des Treibers geliefert. Übrigens ist ein Treiber (»*.DRV«) auch nur eine DLL. Welche Drucker der Anwender installiert hat und welcher aktiv ist, finden wir in der Textdatei »WIN.INI«, die Windows bei jedem Start interpretiert und ausführt, die sich aber auch zur Laufzeit aktualisieren läßt. Mittels einer Windows-Funktion sucht und lädt man diese Zeile, zerlegt sie in ihre drei Worte, hängt (an ein Wort) das noch fehlende ».DRV« an und lädt dann die DLL. Mit einer weiteren Windows-Funktion »GetProcAddress()« läßt sich die Dialogfunkion im Treiber finden, wobei unterstellt wird, daß der Druckerhersteller vereinbarungsgemäß deren Einsprung »DeviceMode« oder »ExtDeviceMode« (es gibt manchmal zwei Dialoge) genannt hat. Das Ergebnis aller Mühen ist ein Zeiger auf die Dialogfunktion. Diese Funktion wird mit einer Struktur und noch einigen Argumenten aufgerufen. Wenn der Anwender den Dialog schließt, kann man seine Wahl in der Struktur ablesen.

Der Menüpunkt »Info« zeigt, wie ein Dialog aufgerufen wird. Das Problem dabei: Die Dialogfunktion ist eine Callback-Funktion. In unserem Fall heißt das, daß der Dialog voll von Windows verwaltet wird. Nun soll aber die Menüanwahl diesen Dialog auf den Schirm bringen. Deshalb wird mit »MakeProcInstance« eine Dialog-Instanz gebildet und dabei die Adresse der Dialogfunktion ermittelt. Damit und mit dem Ressource-Namen des Dialogs wird »DialogBox()« aufgerufen. Nun ist der Dialog auf dem Schirm und die Funktion »About()« arbeitet. Vor dem Aufbau sendet Windows eine »WM_INITDIALOG«-Message, was hier genutzt wird, um den Dialog zu zentrieren. Ansonsten kann man wieder diverse Messages abfragen. Es wird nur geprüft, ob der User den OK-Schalter anklickt oder <Esc> drückt, was den Dialog beendet. Der Aufrufer kann die in globalen Variablen übermittelten Ergebnisse auswerten, um dann mit »FreeProcInstance()« die Instanz freizugeben.

Alle Menüpunkte des Bearbeiten-Menüs zeigen ein interessantes Feature von Windows auf. Ein kompletter Editor mit Mausbedienung, Rollbalken und dem ganzen Service der Zwischenablage ist nämlich in Windows schon eingebaut. Deshalb senden alle Menü-Handler nur Kommandos an Windows. Das ist übrigens typisch. Nachdem ein Fenster angelegt wurde, hat man nur noch sein Handle. Will man dann etwas von dem Fenster, greift man in der Regel nicht auf die Struktur zu, sondern verschickt Messages oder ruft spezielle Funktionen auf. Via »EnumProp()« kann man zwar auch auf die Eigenschaften (Properties) eines Fensters zugreifen, doch darf (sollte) man dann nur Properties ändern oder löschen, die man vorher selbst hinzugefügt hatte.

Ganz nett ist auch die Message »WM_QUERYENDSESSION«. Diese sendet Windows an alle Tasks, wenn der Anwender Windows verlassen möchte. Jeder Task sollte dann seine Endrunde drehen und schließlich »TRUE« zurückgeben. Tut ein Programm das nicht, ist es nicht möglich, Windows zu verlassen.

Die weiteren Messages machen klar, was noch zu tun ist. So etwa sollte ein Editor-Window nach einer »WM_SETFOCUS«-Message (das Fenster wurde aktiviert) den Caret-Cursor (das Schreibmarkensymbol) setzen. Wichtig ist jedoch in dieser Applikation, auf Größenänderungen des Fensters (WM_SIZE) zu reagieren. Hier trifft man auch den relativ häufigen Fall an, daß in »lParam« (einem Langwort) zwei Daten in den beiden Wörtern stehen. »lParam« wird aber auch sehr häufig ein Zeiger sein.

Bleibt als Fazit, daß die Kunst der Windows-Programmierung darin liegt, von über 600 Funktionen und Hunderten von Messages die richtigen zum richtigen Zeitpunkt einzusetzen. Alles, was mit der Oberfläche zu tun hat, lösen CASE-Tools und Ressource-Editoren fast automatisch.

Alle Listings finden Sie auch auf unserer Diskette zum Heft (Seite 114). Selbstverständlich sind die Windows-Programme nicht mit den gängigen Amiga-C-Compilern zu übersetzen. Da muß schon Windows und ein entsprechendes Tool her. *rz*

```
 1: switch (Message)
 2:   {
 3:
 4:     case WM_INITMENU: // Wenn Menü aufgebaut wird:
 5:          // Nicht verfügbare Menüs abblenden, z.B. "Paste",
 6:          // wenn kein Text in der Zwischenablage steht.
 7:          if(OpenClipboard(hEditWnd))
 8:          {
 9:            if(IsClipboardFormatAvailable(CF_TEXT))
10:              EnableMenuItem(wParam, IDM_B_PASTE, MF_ENABLED);
11:            else
12:              EnableMenuItem(wParam, IDM_B_PASTE, MF_GRAYED);
13:            CloseClipboard();
14:          }
15:          break;
16:
17:     case WM_COMMAND:     // Window-Kommando
18:
19:          switch (wParam)  // Bearbeite Menüs
20:            {
21:              case IDM_D_NEU: // Neue Datei
22:                   /* Wenn vorherige Datei ungesichert,
23:                   Frage ob sichern. */
24:                   if (!QuerySaveFile(hWnd))
25:                     return (NULL);
26:                   bChanges = FALSE; // Nichts geändert
27:                   FileName[0] = 0;  // Kein File-Name
28:                   // Lege neuen Edit-Puffer an:
29:                   SetNewBuffer(hWnd, NULL, Untitled);
30:                   break;
31:
32:             case IDM_D_OEFFNEN:
33:                   // Teste, ob aktuelle Datei gesichert
34:                   if (!QuerySaveFile(hWnd))
35:                     return (NULL);
36:                   // Zeige Standard-Dialog, bewerte User-Wahl,
37:                   // lese Datei ein.
38:                   break;
39:
40:             case IDM_D_SICHERN:
41:                   // Wenn Filename leer, gehe zu "Sichern unter"
42:                   if (!FileName[0])
43:                     goto saveas;
44:                   // Wenn File geändert, speicher ihn
45:                   if (bChanges)
46:                     SaveFile(hWnd);
47:                   break;
48:
49:             case IDM_D_SICHERNUNTER:
50: saveas:           // Zeige Standard-Dialog, bewerte User-Wahl,
51:                   // lese Datei ein.
52:                   SaveFile(hWnd);
53:                   break;
54:
55:             case IDM_D_DRUCKEN:
56:                   // Teste, ob Drucker verfügbar
57:                   hPr = GetPrinterDC();
58:                   if (!hPr)
59:                     { /* Meldung und Abbruch*/ }
60:                   // Definniere  Abort-Dialog.
61:                   // Beschaffe Auflösung einer Druckerseite
62:                   // und Zeilenhöhe. Rechne Anzahl Zeilen/Seite.
63:                   // Lege kompatiblen DC an.
64:                   // Drucke via TextOut().
65:                   // Letzte Seit vorschieben und Ende
66:                   // Abort-Dialog abbauen
67:                   // Speicher freigeben
68:                   break;
69:
70:             case IDM_D_DRUCKERINSTALLATION:
71:                   /* Beschaffe Druckerkontext. Funktion
72:                      ermittelt auch Druckerdaten.
73:                   */
74:                   // Lade Druckertreiber (Name in WIN.INI)
75:                   /* Ermittle Zeiger auf Druckerinst.-Funktion
76:                      und rufe sie auf */
77:                   lpfnDevmode = GetProcAddress(hLibrary,
78:                                              "EXTDEVICEMODE");
79:                   nResult = (*lpfnDevmode) (hWnd, hLibrary,
80:                                            &devmode,
81:                                            (LPSTR) szDevice,
82:                                            (LPSTR) szOutput,
83:                                            NULL, NULL,
84:                                            DM_COPY | DM_PROMPT);
85:                   break;
86:
87:             case IDM_D_INFO: // Zeige About-Dialog
88:                   lpfnProcAbout = MakeProcInstance(About, hInst);
89:                   DialogBox(hInst, "AboutBox", hWnd,
90:                             lpfnProcAbout);
91:                   FreeProcInstance(lpfnProcAbout);
92:                   break;
93:
94:
95:             case IDM_D_BEENDEN:
96:                   QuerySaveFile(hWndMain);
97:                   PostMessage(hWnd, WM_CLOSE, 0,0);
98:                   break;
99:
100:            // Das ganze Bearbeiten-Menü eines Editors
101:            // kann Windows erledigen
102:            case IDM_B_UNDO:
103:                  SendMessage(hEditWnd, WM_UNDO, 0, 0L);
104:                  break;
```

```
105:
106:        case IDM_B_COPY:
107:            SendMessage(hEditWnd, WM_COPY, 0, 0L);
108:            break;
109:
110:        case IDM_B_CUT:
111:            SendMessage(hEditWnd, WM_CUT, 0, 0L);
112:            break;
113:
114:        case IDM_B_PASTE:
115:            SendMessage(hEditWnd, WM_PASTE, 0, 0L);
116:            break;
117:
118:        case IDM_B_ALL:
119:            SendMessage(hEditWnd, EM_SETSEL, 0,
120:                        MAKELONG(0, 32767));
121:            break;
122:
123:        case IDM_B_CLEAR:
124:            SendMessage(hEditWnd, WM_CLEAR, 0, 0L);
125:            break;
126:
127:        case IDM_B_DEL:
128:            OpenClipboard(hEditWnd);
129:            EmptyClipboard();
130:            CloseClipboard();
131:            break;
132:
133:         case IDM_B_FIND:
134:             // Läuft via Standard-Dialog
135:             break;
136:
137:
138:         case IDM_B_REPLACE:
139:             // Läuft via Standard-Dialog
140:             break;
141:        default:
142:            return DefWindowProc(hWnd, Message,
143:                                 wParam, lParam);
144:
145:      } // End of switch wParam
146:
147:
148:   // Weitere Windows-Messages
149:   case WM_QUERYENDSESSION:
150:      /* Wenn Windows beendet wird und
151:         ein File ungesichert, Frage ob OK. */
152:      return (QuerySaveFile(hWnd));
153:
154:   case WM_SETFOCUS: // Window hat Fokus erhalten
155:            // Setze Caret-Cursor
156:            break;
157:
158:      case WM_SIZE: // Größe neu
159:            // in lParam steht neue Width u. Height
160:            MoveWindow(hEditWnd, 0, 0, LOWORD(lParam),
161:                       HIWORD(lParam), TRUE);
162:            break;
163:
164:      case WM_PAINT:
165:            // Clear Background
166:            memset(&ps, 0x00, sizeof(PAINTSTRUCT));
167:            hDC = BeginPaint(hWnd, &ps);
168:            SetBkMode(hDC, TRANSPARENT);
169:            EndPaint(hWnd, &ps);
170:            break;
171:
172:      case WM_CLOSE:
173:            DestroyWindow(hWnd);
174:            if (hWnd == hWndMain)
175:              PostQuitMessage(0);
176:          break;
177:
178:      default:
179:            return DefWindowProc(hWnd, Message, wParam, lParam);
180:   }
181:
182:
183:
184: BOOL FAR PASCAL About(HWND hDlg, WORD Message,
185:                      WORD wParam, LONG lParam)
186: {
187:    switch (Message) {
188:      case WM_INITDIALOG:
189:            cwCenter(hDlg, 0);
190:            return TRUE;
191:
192:      case WM_COMMAND:
193:            if (wParam == IDOK || wParam == IDCANCEL)
194:              return EndDialog(hDlg, TRUE);
195:            break;
196:      }
197:    return FALSE;
198: }
199:
      © 1993 M&T
```

Listing 4: Die berühmte Switch-Orgie zum Message-Handling in einem Windows-Programm sieht man noch häufig

Profitips von Programmierern

Effiziente Assembler-Programmierung

Assembler ist heute eine weit verbreitete Technik, Programme für den Amiga zu entwickeln. Fast alle Spiele werden in Assembler geschrieben, da nur so höchste Geschwindigkeit und kurze, prägnante Algorithmen mit optimaler Ausnutzung des Prozessors möglich sind. Hier bekommen Sie Informationen aus erster Hand – vom Entwickler des schnellen OMA-Assemblers.

von Dietmar Heidrich

Die Assembler-Programmierung erlebt auf dem Amiga ihren zweiten Frühling. Sie ist zwar nicht so bequem wie Hochsprachen, erlaubt jedoch optimale Ausnutzung der Prozessorleistung. Selbst größere Projekte wie z.B. der optimierende Makro-Assembler »OMA« wurde in Assembler implementiert.

Allerdings handelt man sich bei der Assembler-Programmierung eine Reihe Nachteile ein. Die Entwicklungszeit für Assembler-Programme liegt etwa um den Faktor fünf höher als die der Hochsprachenprogrammierer. Fehler treten viel häufiger auf, da sie sich meist erst zur Laufzeit bemerkbar machen und in den wenigsten Fällen schon vom Assembler gefunden werden können. Typprüfungen finden z.B. überhaupt nicht statt. Ein beliebter Fehler ist auch das Überschreiben eines Registerinhalts in Unterroutinen (sog. Seiteneffekte).

Im allgemeinen neigen Assembler-Programmierer dazu, die Zeit für Optimierungen zu verschwenden, die die Gesamtlaufzeit fast gar nicht beeinflussen. Was kann man also tun, um die Assembler-Programmierung zu erleichtern, einfacher und vor allem weniger fehleranfällig zu machen?

Der erste Punkt ist die Verwendung symbolischer Namen. Ein Assembler interpretiert Werte nicht nur als Zahl, sondern auch als Symbol. Sicherlich ist

```
move.l gg_SpecialInfo(a0),a0
```

leichter verständlich als

```
move.l $22(a0),a0
```

Zudem erübrigt sich im ersten Fall ein Kommentar. Wirft man einen kritischen Blick auf die in Zeitschriften veröffentlichten Listings, fällt auf, daß im Quelltextanfang immer wieder Konstanten des Amiga-Betriebssystemes definiert werden. Tut man das nicht – der Fehlerteufel läßt grüßen. Alle gängigen Assembler ermöglichen es, mit Hilfe der INCLUDE-Direktive Include-Dateien zu lesen, in denen wichtige Konstanten vereinbart werden. Das ist auch für die Programmierer interessant, die direkt die Hardware adressieren: symbolische Bezeichner für die Custom-Chips sind in den Includes vorhanden und lassen sich auch hier verwenden. Beispielsweise kann man das BltSize-Register so ansprechen:

```
move  d0,_custom+bltsize
```

Der Aufruf von Betriebssystemroutinen ist ein weiterer Punkt. Obwohl die Include-Datei »exec/libraries.i« die »CALLLIB«- und »LINKLIB«-Makros bietet und weitere in der 2.0-Include-Datei »exec/macros.i« definiert wurden, bieten diese keinen besonderen Komfort, denn der Library-Offset der Systemroutine wird immer noch aus der »amiga.lib« abgeleitet. Der OMA-Assembler V2.05 enthält Include-Dateien fürs Betriebssystem 2.0 und 1.3, in denen die Offsets aller Betriebssystemroutinen definiert sind. Damit erübrigt sich das Linken mit der »amiga.lib«.

Um die Routinen nun auch bequem aufrufen zu können, bieten sich diese Makros an:

```
CALL  .  MACRO
         IFEQ     NARG-2
         move.l   \2,a6   ; A6 enthält noch nicht
                          ; den richtigen Wert
         ENDC
         IFLE     NARG-2
         jsr      _LVO\1(a6)
         ELSE     FAIL CALL MACRO: PARAMETERFEHLER
         ENDC
         ENDM
```

Die Benutzung des Makros ist einfach:

```
         CALL     OpenLibrary
```

falls A6 bereits _SysBase enthält, oder

```
         CALL     OpenLibrary,_SysBase
```

sofern A6 einen anderen Wert hat. Soll ein Aufruf am Ende einer Unterroutine stattfinden, kann man das »jsr« durch ein »jmp« ersetzen und das neuen Makro »CALLRTS« taufen. Das hat allerdings den Nachteil, daß man beim Tracen im Debugger diese Routine dann durchlaufen muß; außerdem kann die Unterroutine dann keine Register restaurieren.

Wichtig ist zudem eine vernünftige Form des Quelltextaufbaus. Die Unterroutinen sollten immer ausführlich kommentiert sein, so daß man auch nach längerer Zeit noch in der Lage ist, den Sinn und Zweck einer Unterroutine und ihre Arbeitsweise zu verstehen. Ein möglicher Header wäre:

```
;*************************************
; Finde ein Zeichen in einem String, der
; nicht mit einem Nullbyte endet.
; Ein: A0: Stringzeiger.
;      D0: Stringlänge.
;      D1: zu findendes Zeichen.
; Aus: D0: Position des gefundenen Zeichens.
;          -1, wenn nicht gefunden.
FindeZeichen
        ...
        rts
```

Das ist natürlich nur ein Beispiel, zeigt aber sehr deutlich, welche Register die Routine für die Parameterübergabe verwendet. Die Beschreibung ermöglicht es, die Routine zu verwenden, ohne ihre innere Arbeitsweise zu kennen.

Richtige Makros erleichtern die Programmierung

Wichtig ist dabei, daß verwendete Register am Anfang gerettet und nach Ende auch wieder restauriert werden, um zerstörte Registerinhalte zu vermeiden. Das funktioniert z.B. mit dem MOVEM-Befehl:

```
FindeZeichen
        movem.l d1-d3/a0-a1,-(SP)
        ...
        movem.l (SP)+,d1-d3/a0-a1
        rts
```

Es muß hierbei darauf geachtet werden, daß das Rückgaberegister D0 nicht gerettet wird, denn dann bekäme man immer die Stringlänge als Ergebnis. Nun ist es unbequem und fehlerträchtig, wenn die Registerliste zweimal angegeben wird. Sind die Listen beim Retten und Restaurieren nämlich unterschiedlich, ist der Stack falsch, und der Rücksprung erfolgt in zufällige Bereiche.

GOLEM NEWS

1/93

GOLEM COMPUTER

TURBO II

Motorola 68030 (MMU) mit 68882 taktbar bis 50 MHz. 2 MB Ram aufrüstbar bis 24 MB "On-Board". Prozessor und Coprozessor getrennt taktbar. Volle Burstmode-Unterstützung. Mit 64-Ram-Interface so schnell wie nie zuvor. Gesockelte 030 und 882, daher nachträglich höher taktbar.

16 MHz	1099.-		
20 MHz	1199.-		
25 MHz	1399.-		
28 MHz	1499.-	40 MHz	P.a.A.
33 MHz	1799.-	50 MHz	P.a.A.

2 MB Nachrüstsatz 250.-
8 MB Nachrüstsatz 600.-

RamBoards

A2000		A1000	
0 MB	99.-	0 MB	149.-
2 MB	249.-	2 MB	299.-
4 MB	399.-	4 MB	449.-
8 MB	699.-	8 MB	749.-

HardDisks

2,5" Zoll AT-Bus Festplatten
inclusive Kabelsatz
Quantum Drives

40 MB	449.-
80 MB	599.-
120 MB	899.-



NEU!!! NEU!!!

Turbokarten für Amiga 500/500+

Kickstart-Umschaltung 1.3 - 2.0
incl. Kick 1.3 für A 600 DM 89.-

Golem Computer Vertrieb
Schwanenwall 44
4600 Dortmund 1
Telefon 0231/522192